CATernel

SMP Interactive uKernel

By

Saad Talaat
&
Menna Essa

Table of Contents

Preface:

CATernel project aims to develop a non-portable kernel (as a start) that uses Unix interfaces and APIs. CATernel was initiated at the start as a collaborative project in technical community called [Computer Assistance Team] ;It has been initiated (and still) by two students (Saad Talaat Saad , Menna Sherif Essa) on late 2011.Goal is to develop a limited monolithic kernel which supports IA32 architecture for educational and learning purposes.

Project is planned to support a single architecture as a start and also pass through various types of kernel models. The goal of that progressive development is to keep the kernel operating in all cases. And making it usable at any phase of development ; However, Porting the kernel would be carried out at least after we reach the monolithic model. But our main architecture is the IA32 , The Progressive model of the CATernel project can be considered as a prototype design model since in this case the Exokernel model will be a prototype to final goal.

Iterative prototyping is the used software model. Every project sprint contains the components of the project plan. And since we're using the [research & code] way in developing the kernel, iterative prototyping is the convenient software model to use.

[Prototype One]

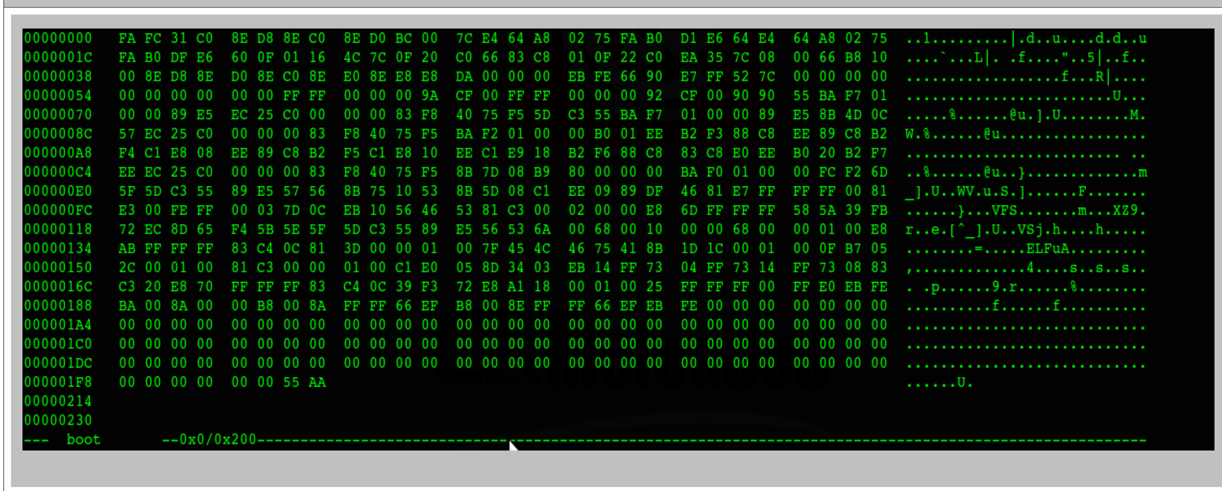
1.The Boot Loader:

1.1 Introduction:

booting is the initial set of operations that a computer system performs when electrical power is switched on ; **A boot loader** is a computer program that loads the main operating system or runtime environment for the computer after completion of self-tests.

When a computer is powered the BIOS comes in control and initializes all data. then it looks for a valid boot loader through in the order of the boot device order. a bootable sector is known by the last 2 bytes in the sector, they must be 0xAA55 (boot signature).That Image has the boot loader of our CATernel.

Note the last 2 bytes



Figure[1-1] The boot loader

When the BIOS find a bootable image it loads the first 512 byte into address 0:07C00 then jump to it. then the boot sector comes in control. it starts execution in the **real mode ,** Real mode is characterized by a 20 bit segmented memory address space (giving exactly 1 MiB[1] of addressable memory) and unlimited direct software access to all memory , I/O addresses and peripheral hardware. Real mode provides no support for memory protection, multitasking, or code privilege levels.

```
.global start
start:
.code16 #since we are in real mode
cli #disable interrupts
cld #clear the direction flag
xorw %ax,%ax #clear the ax register
movw %ax,%ds #clear the data segment register
movw %ax,%es #clear the extra segment register
movw %ax,%ss #clear the stack segment register
movw $start,%sp #set the stack pointer to the bootsector stack
```

Listing 1-1 : Boot sector starts at real mode

1.2 Enabling Protected Mode:

Next it switches to **protected mode** which allows system software to use features such as virtual memory, paging and safe multi tasking.

1.2.1 Enabling A20 Gate:

We start by enabling enabling the A20 Gate [2] for more addressing ; On enabling the A20 gate. first we check if input buffer is full by checking if bit 1 is set on the 0x64 port. then output 0xD1 which makes the next byte passed through the 0x60 port written to IBM AT 804x port. in the next procedure A20.2, you check if the input buffer is full or not , then write 0xDF to the 0x60 port which is written to the IBM AT 804x port which finally enables A20 gate

```
A20.1:
inb $0x64,%al
testb $0x2,%al
jnz A20.1
movb $0xd1,%al
outb %al,$0x64
A20.2:
inb $0x64,%al
testb $0x2,%al
```

```
jnz A20.2
movb $0xdf,%al
outb %al,$0x60
```

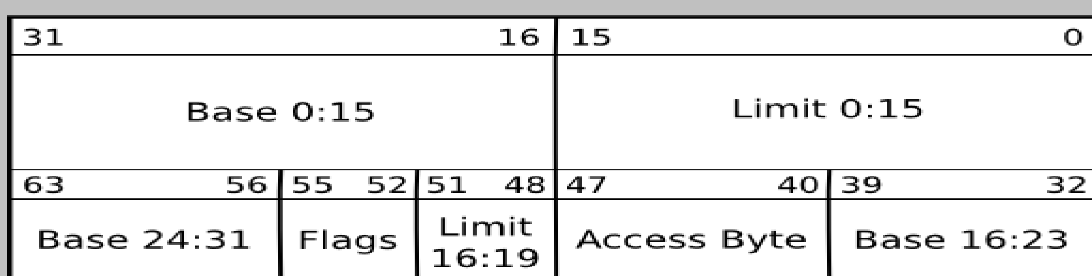
Listing 1-2 : Enabling A20 Gate

1.2.2 Setting up GDT:

Next step is to enable segmentation , For that you need to setup a GDT[3] -Global Descriptor Table- , For the reason that in Protected mode you can't refer to segments simply by doing this :

```
jmp 0000:7c00h
jmp 0002:0020h
```

You cannot directly access segments. That's why a GDT is used , In the GDT is a table where all the segments are defined . yet they are not stored as values but as descriptors. Descriptors has full information about a segment. A Descriptor is 64 bit long.



Access byte is
byte 0 = Accessed bit set to 1 by CPU when segment is accessed. we will set it to 0
byte 1 = read/write permissions
byte 2 = Direction bit we will set that to 0 for growing up segments and 1 for growing down segments and conforming bit
byte 3 = Executable bit 1 if code segment 0 if data segment
byte 4 = always 1
byte 5,6 = Privilege since we are a kernel we will set that to 0

byte 7 = Present bit one for anything

Figure 1-2 : The GDT Descriptor

As in Figure 1-2 , from the first bit till the bit 15, this is the place of the first 16 bits of the segment limit address. and from bit 16 to bit 31 it's the place of the first 16 bits of the segment base address. from bit 32 to 39 more 6 bits of the base address are placed. and then we go into the access bit which is demonstrated above, and the rest of the limit address of the segment. and Flags which is usually set to 0x1100. Finally followed by the rest of the base address. Indexing these descriptors is made by adding 0x8 for every descriptor. So first descriptor's index is 0x0, 2nd descriptor index is 0x8, 3rd descriptor index is 0x10.

```
gdt_table:
.word gdt-gdt_end-1 #gdt table size....mostly 0x17
.long gdt #gdt address
gdt:
.long 0,0 #Null segment
.byte 0xff,0xff,0,0,0,0x9A,0xCF,0 #Code segment
.byte 0xff,0xff,0,0,0,0x92,0xCF,0 #Data segment
```

Listing 1-3 : Setting up GDT Table

After setting up the GDT and loading the gdt address in the gdt register we set the protected mode in cr0 flag. Doing a far jump using ljmp and using the code segment value (0x8) as a segment and next procedure as an offset modified the CS value to the one in our GDT table.

```
switch_mode:
lgdt gdt_table # Load the global descriptor register
mov %cr0,%eax # Load the control register 0 into eax
orl $1,%eax # set the protected mode flag
mov %eax,%cr0 # reset the control register 0
ljmp $CODE_SEG,$protseg #make a far jump to modify the Code
segment
```

Listing 1-4: loading gdt table and switching to protected mode.

After doing this we set the value of data,stack,extra,f,g segments to the one in the GDT. Now we are fully working on protected mode and ready to load the kernel.

```
protseg:
.code32 #since we are working on protected mode
movw $DATA_SEG,%ax #move the data segment value to ax
movw %ax,%ds #set the data segment to data segment value at the
gdt table
movw %ax,%ss # same
movw %ax,%es # same
movw %ax,%fs # same
movw %ax,%gs # same
call cmain # call our kernel loader
```

Listing 1-5 : Setting Registers to the ones in the GDT

1.3 Loading the kernel Image:

Note That we're compiling the kernel as an ELF image , The *cmain* function responsible for loading the kernel is in arch/x86/boot/main.c. What the function is that it reads the ELF file from the Image , loads it into memory at virtual address 0x10000 then jumps to the entry point of the kernel executable which refered to through the kernel's ELF headers (`ELFHDR->entry`) .

To do so it uses to functions , *readsect* and *readseg* in order to read the executable from disk , the two functions are also found in main.c file.

2. Establishing environment.

After kernel is fetched from boot loader, environment is reset and memory thunks and segments are remapped. Several components are set before running the kernel:-

1- Kernel stack.
2- Segmentation and GDT.
3- Page tables and directories.

2.1 Kernel Stack

Since execution stream is now in the kernel(0x100000) and not the boot sector(0x7c00). stack must be reset in order to carry on healthy execution we must reset. Kernel stack is paged size memory thunk which we set to global to be able to refer to once we set up paging.

```
     .p2align  PAGELG         # Will pad allocation to 0x1000
byte
     .global        kernel_stack
kernel_stack:
     .space         KERNEL_STACK
     .global        kernel_stack_end
kernel_stack_end:
```

Listing 2.1 - Kernel stack thunk.

2.2 Segmentation and GDT

while initializing the kernel it is needed to keep in mind the Virtual memory addressing. Thus, Global descriptor table and segmentation is reset to generate new virtual addresses, new kernel segments are 0x10000000 long. Kernel virtual base address is 0xF0000000 which puts the kernel code base address to 0xF0100000. At this point it is important to notice that there's only two segments selectors set.

```
gdt:
	.long 0,0
	SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW|SEGACS_X)	#
code seg
	SEGMENT(0xffffffff,-KERNEL_ADDR, SEGACS_RW)	# data seg
gdt_end:
```

Listing 2.2 - Kernel Initial Global descriptor table.

CATernel adopts a memory map that is not like any *nix system. Kernel is mapped to high memory addresses like windows. However we are not planning to be a UNIX like operating system.

2.3 Page tables and directories

Before setting up kernel the virtual page tables and page directories are set which will be needed for paging later. We shall touch this point later on on a separate chapter.

3.Drivers :

3.1: CMOS/RTC :

CMOS Complementary metal–oxide–semiconductor , including the RTC -Real time clock- is responsible for saving 50 or 114 bytes of setup information for the BIOS , it includes a battery that keeps the clock active.

CMOS is accessed through I/O ports 0x70 and 0x71 , the ports specifications are as follows : note that indexing is little endian [RTL]

```
a)
0070  w     CMOS RAM index register port (ISA, EISA)
             bit 7       = 1  NMI disabled
                   = 0  NMI enabled
             bit 6-0      CMOS RAM index (64 bytes, sometimes 128 bytes)

            any write to 0070 should be followed by an action to 0071
            or the RTC wil be left in an unknown state.

b)
0071  r/w   CMOS RAM data port (ISA, EISA)
            RTC registers:
            00    current second in BCD
            01    alarm second   in BCD
            02    current minute in BCD
            03    alarm minute   in BCD
            04    current hour in BCD
            05    alarm hour   in BCD
            06    day of week  in BCD
            07    day of month in BCD
            08    month in BCD
            09    year  in BCD (00-99)
            0A    status register A
                   bit 7 = 1  update in progress
                   bit 6-4 divider that identifies the time-based
                        frequency
                   bit 3-0 rate selection output  frequency and int. rate
            0B    status register B
                   bit 7 = 0  run
                       = 1  halt
                   bit 6 = 1  enable periodic interrupt
                   bit 5 = 1  enable alarm interrupt
                   bit 4 = 1  enable update-ended interrupt
                   bit 3 = 1  enable square wave interrupt
                   bit 2 = 1  calendar is in binary format
```

```
                    = 0  calendar is in BCD format
                bit 1 = 1  24-hour mode
                    = 0  12-hour mode
                bit 0 = 1  enable daylight savings time. only in USA.
                       useless in Europe. Some DOS versions clear
                       this bit when you use the DAT/TIME command.
               .
               .
               .
```

Listing 3.1 CMOS Ports , a) I/O Port 0x70 , b) I/O Port 0x71 [1]

CMOS values are accessed one byte at a time so we refer to each byte as a CMOS Register, The first 14 CMOS registers access and control the Real-Time Clock.
In port 0x70 -CMOS RAM index register port- as it's name suggests saves the CMOS RAM index which is a Nonvolatile BIOS memory refers to a small memory on PC motherboards that is used to store BIOS settings , Also the last bit in the register indicates whether the NMI -non maskable interrupts-[2] are enabled (0) or disabled (1).

So port 0x70 is used to select the CMOS Register to read , so if you want to read register 0A which holds Status A register you simply do this :
`outb(0xA,0x70);`

For making things readable , we define all the indexes and constants in cmos.h headers

```
/* CMOS Registers */
#define CMOS_INDEXPORT     0x70
#define CMOS_DATAPORT      0x71
/* RTC Registers */
#define    RTC_SECONDS     0x0
#define RTC_ALRMSECOND     0x1
#define RTC_MINUTES   0x2
     .
     .
     .
```

Listing 3.2 cmos.h

From here we can introduce the first CMOS function ,*cmos_get_reg*; which will take the register the needs to be read as an input , offset the index register and read it. to explain the procedure in more details , if you see the code in listing [3-3] you'll see that we first read Status A register if bit 7 is 1 , that is the register = "10000000" = 0x80 , means that the CMOS is being updated and you can't use it now so the function busy waits and keeps reading the register until it's free . Then it offsets the index register to the desired register to read held in the parameter "value" , finally reads it from the Data port(0x71) and return it.

```
uint32_t cmos_get_reg(uint8_t value){
     uint32_t val;
     uint8_t update;
     //check status
     while(update == 0x80){
          outb(RTC_STATUS_A,CMOS_INDEXPORT);
          update = inb(CMOS_DATAPORT);
     }
     cli();
     //get the value
     outb(value,CMOS_INDEXPORT);
     val = inb(CMOS_DATAPORT);
     return val;
}
```

Listing 3.3 cmos.c , reading cmos registers

Next function is *cmos_get_time* which is similar to *cmos_get_reg* which deals only with time -register 0:9- , The function is to be deprecated.

Finally , *cmos_set_power_stat* is responsible for supporting status B register which includes the power options and status It follows the same sequence as the previous functions but sets the register to the values mentioned in status B described in listing [3-4] .

```
/*those 3 sets the respective bit to zero so we mask with AND*/
     if(stat==STAT_RUN || stat==STAT_CAL_BCD ||
stat==STAT_CAL_HR12)
       New_Stat &= stat;
     else
       New_Stat |= stat;
```

Listing 3.4 setting status B power options.

The rest of the registers are not yet used hence not yet supported.

---

[1]: The full detailed prots description can be found at
http://bochs.sourceforge.net/techspec/PORTS.LST

[2]: Refer to interrupts chapter for more details.

3.2 Video

Almost all new kernels and operating systems access and use video mode using the VESA/VBE interface. to interface with the attached video card. however in CATernel we choosed to support legacy and old devices before supporting newer versions, therefore we choosed to use CGA(Color Graphic Adapter) to support video and console. There's few differences between CGA and EGA and VGA.

3.2.1 CGA(Color Graphic Adapter)

As mentioned earlier CGA is an old graphic adapter which we choosed to support first in CATernel. It supports two modes, Text mode and Graphic mode. for now we are only using the Text mode.

3.2.1.1 Text Mode

CGA has two text modes with a fixed character size.

40x25 Mode : Each character is 8x8 dots size and has up to 16 colors with resolution of 320x200.

80x25 Mode : It has the same character size and same color count, but with 640x200 resolution.

on CATernel we will support the CGA in text mode(80x25).the memory storage is two bytes of video RAM used for each character. 1st byte is the character code and the 2nd is the attribute. a screen might be 2000 byte or 4000 byte (40*25*2) , (80*25*2). and CGA's video RAM is 16Kb. and of course all what we can output is ASCII.

```
bit 0 = Blue foreground
bit 1 = Green foreground
bit 2 = Red foreground
bit 3 = Bright foreground
bit 4 = Blue background
bit 5 = Green background
bit 6 = Red background
bit 7 = Bright background (blink characters)
```

Listing 3.5 Character color attributes

To control screen cursor and lines two registered are used, Index and Data registers at address 0x3D4 and 0x3D5 respectively. If you took a look at video.c code you will find setters and getters for position. so position desired to read is supplied to index register which is 0xF for the first byte in the position and 0xE for the second byte.

and since we are working on a 80*25 then we wont need more than 4 bytes. for getting the value i read from the data register after specifying the index I want to read and I *inb* the value coming from the data port. for example:

suppose the cursor position is at 0x5a0, so what you will first get is the first byte of the position which is 0xa0. and as you might have noticed we do no operations on that. but on the second position you get 0x05. the operations is for mixing the first byte and second byte so they would make 0x5a0.

also you may notice that CGA_BUFF_OFFSET. which is the offset of the CGA video RAM in memory.

cga_putc:

what i do here is that i put the character i want to type on the screen to the CGA video RAM. and since we are working on 80*25 resolution bytes after the offset 0xB87D0 wont be written to screen yet they will be written to video RAM. this issue can be handled using memory trick like...move all binaries from 0xB8080 to 0xB87D0 80 byte backward which is the row size in 80*25 resolution. and then move the cursor position 80 place backward.

cga_putstr:

this function passes a pointer to an array of characters which are passed in a loop character at a time till we reach the null terminator character.

CGA Ports:

```
0x3D4     -    index register
0x3D5     -    data register
0x3D6     -    same as 0x3D4
0x3D7     -    same as 0x3D5

0x3D8     -    CGA mode control register
|->  bit 5 - blink register
```

```
|->  bit 4 - 640*200 High-Resolution register
|->  bit 3 - video register (if cleared the screen wont output)
|->  bit 2 - Monochrome
|->  bit 1 - text mode
|->  bit 0 - Resolution

0x3D9     -    Palette Register / Color control register
|->  bit5 - chooses color set
|->  bit4 - if set the characters show in intense
|->  bit3 - intense border in 40*25 and intense background in
     300*200 and intense foreground in 640*200
|->  bit2 - red borders in 40*25, red background in 300*200,
     red foreground in 640*200
|->  bit1 - green border in 40*25, red backround in 300*200,
     red foreground in 640*200
|->  bit0 - blute border in 40*25, red background in 300*200,
     red foreground in 640*200

0x3DA     -    Status register
|->  bit3 - if set then in vertical retrace.
|->  bit2 – light pin switch off
|->  bit1 - positive edge from light pen has set trigger
|->  bit0 – 0 do not use memory.

0x3DB     -    clear light pen trigger
0x3DC     -    set   light pen trigger
```

Listing 3.6 CGA Ports

Console dependency:

The early console depends on the video driver and keyboard. On the console there's several wrapper functions for the video driver like *console_putc* , *putchr* and *console_clear.*

3.3 PS/2 Keyboard

Unlike video driver, CATernel's keyboard driver is similar to every kernel's keyboard support. However, since Keyboard is a serial device It uses almost a unified interface like any other PS/2 device. CATernel keyboard driver supports only one function which is a keyboard interrupt handler, Of course at this point a keyboard interrupt does not occur since CATernel busy wait on any I/O device.

Like any other device Keyboard commands are passed through I/O ports. In CATernel we barely make use of the keyboard controller, the only two operations are made through the status and data port. First operation is to check the keyboard data register, second one is two read that character from the data port.

```
0064 r    KB controller read status (ISA, EISA)
     bit 7 = 1 parity error on transmission from keyboard
     bit 6 = 1 receive timeout
     bit 5 = 1 transmit timeout
     bit 4 = 0 keyboard inhibit
     bit 3 = 1 data in input register is command
           0 data in input register is data
     bit 2     system flag status.
     bit 1 = 1 input buffer full
     bit 0 = 1 output buffer full
```

Listing 3.7 Keyboard Status port

There's three different scan code sets, CATernel uses the first scan code set. a scan code determine what key is pressed and three keyboard maps are provided, the first is a character map on normal case, second is a character map of keyboard on shift case and third is a character map of keyboard once a toggle button is on.

A keyboard interrupt handler reads the scan code and starts determining what key was pressed and then it is returned to interrupt issuer.

Console dependency

Console uses keyboard controller as an input device, a wrapper function called *console_getc* issues a keyboard interrupt and and index that char to a screen position(but char is not printed).

3.4 Intel 8259 PIC(Programmable Interrupt Controller)

As mentioned before on CATernel we tend to support legacy and old devices first, therefore we choosed to support Intel 8259 PIC before APIC(Advanced PIC) and IOAPIC are supported. Interrupts are the only way to manage execusion over x86 machines since Intel is an interrupt driven ISA. on old machines when only real mode was used PIC was the controller for interrupts and interrupts were handled by what is called vector store in an Interrupt vector table. An interrupt vector table is similar to the modern interrupt decriptor table except that IVT has the vectors already stored on the BIOS. PIC consists of two chip (Master/Slave) each has 8 interrupts which makes the total interrupts 16. an PIC interrupt is called IRQ(interrupt request) since an interrupt can be blocked when a higher priority interrupt handler is currently executing.

```
INT# 00 > F000:FF53 (0x000fff53) DIVIDE ERROR ; dummy iret
INT# 01 > F000:FF53 (0x000fff53) SINGLE STEP ; dummy iret
INT# 02 > F000:FF53 (0x000fff53) NON-MASKABLE INTERRUPT ; dummy iret
INT# 03 > F000:FF53 (0x000fff53) BREAKPOINT ; dummy iret
INT# 04 > F000:FF53 (0x000fff53) INT0 DETECTED OVERFLOW ; dummy iret
INT# 05 > F000:FF53 (0x000fff53) BOUND RANGE EXCEED ; dummy iret
INT# 06 > F000:FF53 (0x000fff53) INVALID OPCODE ; dummy iret
INT# 07 > F000:FF53 (0x000fff53) PROCESSOR EXTENSION NOT AVAILABLE ; dummy iret
INT# 08 > F000:FEA5 (0x000ffea5) IRQ0 - SYSTEM TIMER
INT# 09 > F000:E987 (0x000fe987) IRQ1 - KEYBOARD DATA READY
INT# 0a > F000:FF53 (0x000fff53) IRQ2 - LPT2 ; dummy iret
INT# 0b > F000:FF53 (0x000fff53) IRQ3 - COM2 ; dummy iret
INT# 0c > F000:FF53 (0x000fff53) IRQ4 - COM1 ; dummy iret
INT# 0d > F000:FF53 (0x000fff53) IRQ5 - FIXED DISK ; dummy iret
INT# 0e > F000:EF57 (0x000fef57) IRQ6 - DISKETTE CONTROLLER
INT# 0f > F000:FF53 (0x000fff53) IRQ7 - PARALLEL PRINTER ; dummy iret
INT# 10 > C000:014A (0x000c014a) VIDEO
INT# 11 > F000:F84D (0x000ff84d) GET EQUIPMENT LIST
INT# 12 > F000:F841 (0x000ff841) GET MEMORY SIZE
INT# 13 > F000:E3FE (0x000fe3fe) DISK
INT# 14 > F000:E739 (0x000fe739) SERIAL
INT# 15 > F000:F859 (0x000ff859) SYSTEM
INT# 16 > F000:E82E (0x000fe82e) KEYBOARD
INT# 17 > F000:EFD2 (0x000fefd2) PRINTER
INT# 18 > F000:B023 (0x000fb023) CASETTE BASIC
```

Figure 3.1 IVT listed by Bochs

3.4.1 Master and Slave PIC

To handle all the 16 we must be able to index the right interrupt to the right PIC and when an EOI is issued we should be able to determine which PIC should handle the EOI. Master PIC and slave PIC have their own ports, Master has 0x20/0x21 and Slave has 0xA0/0xA1. the functionality is similar except for the interrupt types they handle.

```
1- Intel i8253 PIT
2- Keyboard
3- Video Interrupt
4- Serial port 2
5- Serial port 1
6- Fixed Disk
7- Floppy disk
8- Parallel printer
9- Real time clock (RTC)
10- Cascade Redirect
13- Mouse interrupt
14- Coprocessor exception
15- Primary Hard disk
16- Secondary Hard disk
```

Listing 3.8 PIC Interrupt requests (IRQs)

PIC I/O is a little different that former devices we dealt with in CATerenl. PICs adopt a terminology called ICW(Initialization command word) and OCW(Operation command word). Intel 8259 manual defines ICW that It is used before any normal operation. as for OCW it can be executed at any point after initialization.

```
0020 w    PIC initialization command word ICW1
           bit 7-5 = 0  only used in 80/85 mode
           bit 4 = 1  ICW1 is being issued
           bit 3 = 0  edge triggered mode
                 = 1  level triggered mode
           bit 2 = 0  successive interrupt vectors use 8 bytes
                 = 1  successive interrupt vectors use 4 bytes
           bit 1 = 0  cascade mode
                 = 1  single mode, no ICW3 needed
```

```
           bit 0 = 0  no ICW4 needed
                 = 1  ICW4 needed
```

Listing 3.9 Port 0x20 flags for ICW1

3.4.2 PIC in protected mode

In x86 protected mode interrupts are only handled by the IDT and the IVT is omitted. such a case will put us in a problem whenever an IRQ is issued since it will conflict with Intel default 0~32 exceptions, therefore we won't be able to distinguish an exception from an IRQ. Luckily, Offsetting the IRQ indexes is a functionality can be performed through PIC ICWs. This is done through ICW2 in particular.

3.4.2.1 Initializing PICs

PIC initialization is done once the kernel have reached protected mode main-flow execution. It enables IRQs to be handled using IDT after offsetting them so CATernel would be able to use PIC in protected mode. this is done on four steps

First Step, ICW1 is passed a value with Flags ICW4 needed and ICW1 issued flags. Second Step, ICW2 is passed a value with the base offset desired to IRQs, and Since we do this step for both PICs the slave PIC base offset is passed as master PIC offset plus 8. Third Step, ICW3 is passed a value that holds one shifted by the interrupt pin of the slave PIC. and slave PIC ICW3 takes slave PIC index. Fourth Step, ICW4 takes a value with 8088/8086 mode flag set. However other flags are also active (0/1).

3.4.3 Masked Interrupts

Masked interrupts is another terminology an i8256 adopts, since it supports enabling and disabling interrupts through setting masks holds flags of desired interrupts and undesired. an interrupt could be disabled by setting its corresponding flag. Interrupt masks are held in a PIC register called IMR(Interrupt Mask Register). each PIC has its own IMR since each PIC has its own type of interrupts.

```
0021 r/w  PIC master interrupt mask register
          OCW1:
           bit 7 = 0  enable parallel printer interrupt
           bit 6 = 0  enable diskette interrupt
           bit 5 = 0  enable fixed disk interrupt
           bit 4 = 0  enable serial port 1 interrupt
           bit 3 = 0  enable serial port 2 interrupt
           bit 2 = 0  enable video interrupt
           bit 1 = 0  enable keyboard, mouse, RTC interrupt
           bit 0 = 0  enable timer interrupt
```

Listing 3.10 IMR in Master PIC

Such a functionality gives the ability to disable the whole PIC by setting all flags on both master and slave PICs.

3.4.4 EOI End Of Interrupt

EOI must be used by an IRQ handler since it notifies the PIC that issued an interrupt that the interrupt handler has finished its execusion, so the PIC should insert its blocked interrupt (if exists) to processor.

```
0020 w    OCW2:
           bit 7-5 = 000 rotate in auto EOI mode (clear)
                = 001    nonspecific EOI
                = 010    no operation
                = 011    specific EOI
                = 100    rotate in auto EOI mode (set)
                = 101    rotate on nonspecific EOI command
                = 110    set priority command
                = 111    rotate on specific EOI command
           bit 4     = 0 reserved
           bit 3     = 0 reserved
           bit 2-0  interrupt request which the command applies
```

Listing 3.11 EOI using OCW2

3.5 Intel 8254 PIT(Programmable Interrupt Timer)

Due to the problem with RTC periodic interrupts [Appendix A] we had to support another device for time slicing execusion. The next modern device is PIT but still PIT is a legacy device. however, PIT was easier to program that RTC.

PIT has three channels. First channel is for counter divisor, second channel is for RAM refresh counter and third one is for issuing a beep on cassette or speaker on an interval. For the kernel clock we shall only use the first one which is the divisor of the frequency of the PIT to issue on interrupt on a subsequent interval. in CATernel we set this to issue 20 interrupt per second and handle those interrupts by scheduler.

4. Memory Management

On kernel level, Memory management and allocation is a very crucial and critical part that composes an efficient performance and protection. Dealing with memory on kernel level has to be very careful and clever since kernel doesn't have the user space luxuries like memory allocation errors.

Unlike Linux, BSD and Spartan kernels, CATernel doesn't yet contain any *Zone terminology* although there's different thunks of memory CATernel doesn't make use of the whole memory. Only two thunks of memory are used, Base and extended memory. and allocation on kernel level is done by manually allocating a memory unit which here is Paging.

4.1 Paging

A page is the smallest unit of memory in CATernel. Although a page is considerably huge comparing to processor smallest unit of memory which is a byte the Intel MMU deals with pages as the basic unit of virtual memory. IA32 has different types of paging methods called "Paging Modes", First is 32-bit paging which addresses 32-bit physical addresses to 32-bit virtual addresses, Second is PAE Paging which is used to translate 52-bit Physical addresses to 32-bit Virtual addresses and the third mode is called IA32e Paging which is used to translate same size of former physical addresses to 48-bit virtual addresses.

Since we're applying a minimal implementation for paging in CATernel we are only making use of 32-bit paging which has two types of daat structures Page tables and page directories and two modes each has a different page size.

4.1.1 32-bit Paging structures

Two types of Data structures exist to index a page in 32-bit mode. It can be considered a two dimensional page array with the higher level is the page directory. Page directory contains page tables addresses and several flags, each member of page directory is called PDE(Page Directory Entry). Page table contains the addresses of physical pages and serveral flags, each member of page table is called PTE(Page Table Entry).

```
a 32-bit KByte Paging PDE would be like this.
          [0,11] Page Table Permissions and Ignored bits.
          [12,31] Page Table address.

a 32-bit KByte Paging PTE would be like this.
          [0,11] Page permissions
          [12,31] Page physical address

the permissions of the PDE is almost the same of the
permissions of the PTE. PDE permissions:
          [0]  First bit must be always 1 which marks page as
          (Present), if it's not set the entry is ignored.
          [1] R/W permissions, If bit is cleared no write
          operations are allowed
          [2]  U/S, it indicates whether the page/page table
          belongs to user of supervisor (ring 0,3). if it's
          cleared, it means it belongs to supervisor which
forbids access from CPL =3
          [3] WT, Write through flag it indicates the memory
type used to access this page either
               write back caching or write through caching.
          [4] CD, it indicates if this page is cachable or not,
if set it's not cachable.
          [5] A, Accessed flag refers to whether a software
accessed this page or not.
          [6] D, Dirty flag is set if a software did a write
operation to this page.
          [7] PS, Page size flags if set it means we're using
4MByte paging, if not it's 4KB paging
          [8] G, if set it means that the directory translation
is Global. we shall refer to it later.
```

Listing 4.1 Paging structures entries

4.1.2 32-bit Paging Modes

First is 4Kbyte page 32-bit Paging, It uses two data structures to index a page (Page Directory - Page Table), Second type is 4Mbyte page 32-bit paging, which uses only one data structure to index a page (Page Directory). In CATernel the first mode is used since it will enable smaller pages hence, smaller basic memory units. A smaller memory unit has some advantages and disadvantages. having a 4Kbyte page as a virtual memory smallest unit will provide less fragmentation. on the other hand, a bigger memory size will be used to store paging data structures.

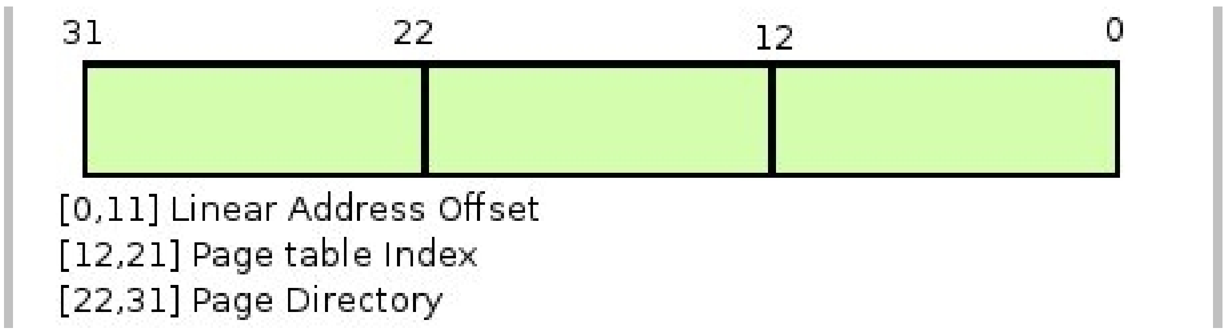


Image 4.1 Linear address in paging

A linear address in 4Kbyte 32-bit paging mode contains three fields, Linear address offset, Page table entry index and page directory entry index. in 4KByte Paging, and since we refer to 4Kbyte sized page which is 2^13 it means we can offset with FFF into the page. from 0xFFFFF000 to 0xFFFFFFFF for example. the other [12,21] bits indicate the index of the page in the page table.

4.1.3 Initializing paging

Allocating/Clearing Page Directory

we use the boot time allocation scheme to allocate 4096 bytes of memory right after the kernel LOAD segment and clear it. and to provide access to page table for both user and supervisor to access the page directory by making it recursively reference itself when a virtual page table address is used. in our case, VIRTPGT, USERVIRTPGT. which lie in 0xEFC00000 , 0xEF400000 repectively. So for those page numbers/linear address to refer to page directory itself we map it to itself by this line.

```
pdr[PGDIRX(VIRTPGT)]=KA2PA(VIRTPGT)|PAGE_PRESENT |PAGE_WRITABLE;
```

and the index of this entry is 3BF. it looks in bochs like this.

```
<bochs:5> x/10x 0xf010befc
[bochs]:
0xf010befc <bogus+       0>:    0x0010b003      0x03ffd027      0x03ffc007      0x03ffb007
0xf010bf0c <bogus+      16>:    0x03ffa007      0x03ff9007      0x03ff8007      0x03ff7007
0xf010bf1c <bogus+      32>:    0x03ff6007      0x03ff5007
<bochs:6>
```

Image 4.2 recursive page directory indexing

Pages data structure

What first comes in your mind if you need to detect whether there's a free page or not is to scan the page directory and table and detect free pages and search whether the page you want to map lies between those free pages or not. this would create a MASSIVE overhead. But a better way to do this is to create a linked lists, of Pages structures or whatever it might be called, it's not actually page structures but it's a (struct Page list) this is a simple backward linkedlist entry with a pointer to previous element and a value field, in our case this field is called ref which indicates how many pointers or procs refer to that page of course if it's allocated. if this ref field is 0 it makes this page free to use. and this is how it looks in memory.

```
<bochs:19> x/16x 0xf010d000
[bochs]:
0xf010d000 <bogus+       0>:    0x00000000      0xf010d00c      0x00000001      0x00000000
0xf010d010 <bogus+      16>:    0xf010d018      0x00000000      0xf010d00c      0xf010d024
0xf010d020 <bogus+      32>:    0x00000000      0xf010d018      0xf010d030      0x00000000
0xf010d030 <bogus+      48>:    0xf010d024      0xf010d03c      0x00000000      0xf010d030
<bochs:20>
```

Image 4.3 pages list

Initializing structures.

after setting up the environment, we start to initialize page directories and tables. and since we need to be still operating after paging activation we need to put entries for both Kernel code and stack so after paging is active the same addresses would be translated to same physical position.

in steps,

we map the whole memory into pages and start filling out the free pages list. this can be done by a loop. but there's a memory we need to mark used that has the ACPI system calls and Memory mapped I/O [Section 6.1] plus the kernel code/stack segments are also in use, so those we need to mark used as well. after filling the free pages list. memory mapping procedure should be supported to map physical segments to virtual addresses in runtime. This is the map_segment_page function. to provide such a function we need other functionalities, Like the ability to find and create a page table at a specific position, and insert or remove a page. and allocate a page. a simple page allocation and freeing functions is to simply remove and add a page member to the free pages global list.

to insert and remove pages you need to be able to locate and create new Page dir Entries that are dependent on the virtual address. for this x86_pgdir_find function is defined, it takes the virtual address which's PDE is desired to be found or created. the function first checks if this entry already exist, if yes a PDE is returned. (it is refered in the code as PTE since it references the table, however it's called PDE in intel manuals) if not it is created if desired.

to remove or insert a page directory entry to a page directory other two functions were defined, removal function uses a lookup function to determine the existance of PDE, then it executes a detach funtion that determine whether there's processes that are still using this page or not, if not page is freed. then the pte is set to NULL or 0 and TLB is updated. to insert a page, the function first checks if there's a PDE/PTE refering to this page or the creatability of PDE/PTE refering to this page, if the page already is refered. if yes it's freed and reallocated. but we won't be using these functions ATM.

At this point mapping a segment of memory to virtual memory is trivial, for each page of the segment you insert a PDE and and a PTE that refer to this VA and an opposing PA, this makes the PTE random.

4.1.4 Triggering paging

to activate paging we need to first map the pages array so we can still access it via the same virtual address, also we need to map the kernel stack and the kernel code. a kernel code mapping for instance looks like this:-

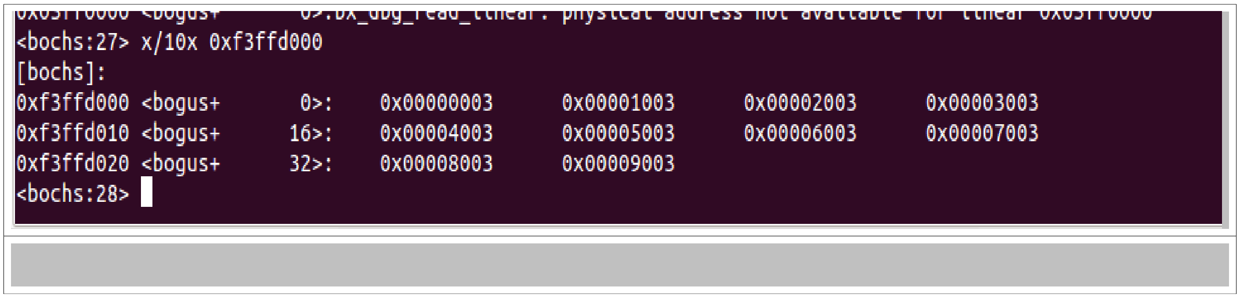
```
map_segment_page(pgdir,KERNEL_ADDR,0x10000000,0,PAGE_PRESENT|PAGE_WRITABLE);
```

this maps virtual address 0xF0000000 to 0xFFFFFFFF to the pages from 0 to 0xFFFFFFF. let's do this manually we're not treating the 0xF0000000 anymore as segment base, but as a page number. which's PDE Index = 3C0 which means the entry's offset from pgdir base is 3C0*4 = F00, let's check that in the bochs debugger.

```
<bochs:24> x 0xf010bf00
[bochs]:
0xf010bf00 <bogus+       0>:    0x03ffd027
<bochs:25>
```

Image 4.4 PDE in memory

It is noticable that the PDE is marked Accessed since it's already executing. now let's read the Page table, the 0x03ffdXXX refer to the physical address of page table, and since it's in the 0 ~ 0x10000000 kernel addr ess space we'll just add a FXXXXXXX to it.



as you see pages are sequentially ordered in page table as PTEs.

eventually, we load page directory address to cr3 and trigger paging on CR0. after we set the first PDE as the Kernel codes PE. since after paging the 1st PDE is loaded. then we reset segment table to full 4GB memory since we're able to convert 4GB of linear addresses to physical addresses. and a far jump is done to the same code it preserve the execution of the code after removing the page directory[0]. and for the CS to get updated.

4.1.3 Allocation

CATernel uses a weak memory allocation schemes, First is used on boot time which allocates memory heaps after kernel code section. Second is used by paging manager which allocates one page at a time. However, that's a subject to be looked in later, Zoning and slab allocators might be used.

4.2 Segmentation

In CATernel we use segmentation effectively on pure segmentation. and we only use the Global descriptor table. We use segmentation in re-mapping the kernel physical address into a virtual address, and maintain the 32-bit addresses. A global descriptor table is used for the OS level/ring 0 to locate its Code/Data/TSS segments. other tasks use the Local descriptor Table [LDT] which differs from a task to another. Like *Nix and WinXP we only use Paging to have protection and addresses virtualization. once paging is activated, we set GDT selectors base address to 0 with MAX memory limit. since we don't need longer addresses like protected mode. But, It's intended to have full power of Intel memory management like in OS/2.

5. Interrupts and system calls

5.1 Interrupts:

According to Intel Software developer manual Vol 3 [Chp.6 Interrupts and Exception Handling] an Interrupt can be defined as follows:-
Interrupts occur in random times during execution, and they are invoked by hardware. Hardware uses interrupts to handle events that are external to the processor such as request to service a device. software can generate an interrupt also by instruction INT <interrupt_number> also according to varying sources interrupt numbers break into the following.

-IRQ [Interrupt ReQuest]:
Interrupt requests are from 0 to 16

-Interrupts:
Interrupts are from 0~31, 8~16, 70h~78h

-PIC/Keyboard Ports
Programmable Interrupt controller and keyboard ports.

an Exception can be defined as follows:-

> Exception occurs when a processor detects an error during trying to execute an instruction like devision by 0. the intel processor detects many error conditions including protection violation as page faults.

0	Programmable Interrupt Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1 / Unreliable "spurious" interrupt (usually)
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

Listing [5-1]Standard ISA IRQs

Int	Description
0-31	Protected Mode Exceptions (Reserved by Intel)
8-15	Default mapping of IRQ0-7 by the BIOS at bootstrap
70h-78h	Default mapping of IRQ8-15 by the BIOS at bootstrap

Listing [5-2] Default PC Interrupt Vector Assignment

5.1.1 Dummy interrupts

For the sake of prototyping, There was a dummy exception and interrupts handler created under a generic name for both as Interrupt. However this still stands till

now, but few modifications was mode which will be mentioned later on the document.
an interrupt occurrence causes an interrupt handler to be executed from the protected mode Interrupt Descriptor Table. the indexed function that matches interrupt number will be executed. To do that we must first initialize the IDT by filling the first 64 interrupt by a semi interrupt handler and filling the rest 196 interrupt with a dummy iret. Once an interrupt occur the execution is altered to kernel space and starts executing the interrupt handler. Before an interrupt handler executes by default there's a stack frame storing the previous cpu state (the interrupt issuer) to be able to return to it, however this is not enough for us. In CATernel we define a soft cpu state structure holding all general purpose and segment registers and address space of the current environment.

Since there's no interrupts (not exceptions) that are yet handled through CATernel except for RTC periodic interrupts and system calls, any other interrupt once issued from user it goes through the interrupt mapping function and returns to user if no handler exists. yet there's illegal interrupts to be issued by user like the RTC which is used in scheduling.

5.1.2 Back end

-The gatedesc structure :
Our initial gate descriptor structure is more like the one in HelenOS, although i find it to be almost useless, since We won't be really using the args, reserved. And it's time consuming executing an assignment statement for every member of type,dpl and present bit. such a structure is implemented in both Linux/Minix as a one type_dpl_present field. We shall save this for later.

-CPU state frames :
to provide an informative and effective switching between caller and interrupt vector, cpu state frame holds info about variants of caller environment. such a frame in Minix for instance holds(vector, error code, eip, cs, eflags, esp, ss) in Linux all registers exist which is the same as HelenOS, although order is different.

5.1.3 Page faults

To this point we have a minimal user space, and poor scheduling mechanism and no signaling since we busy wait on resources. This simplifies the page fault handling for us in CATernel. Page fault handler checks if a page fault came from kernel mode, If so kernel is paniced. If it came from user space by user trying to access kernel space, the exception issuer proc is killed. If a page fault is done by user by jumping to wrong address (by instruction fetch) issuer proc is killed. if the issuer proc did issue a page fault by exceeding the stack, the proc stack is increased if it didn't reach max size for a user space stack.

5.2 System calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O.users do not need to concern themselves with system calls as they will be all done virtually in a library .

for a system call to happen a number of steps are followed:

|user code|->|Intermediate Library|->|Kernel code|

First , The program calls the function in the user library , this function is responsible for indexing and passing the arguments of the
kernel function , then it issues and SYSCALL(0x30) interrupt

Before going into code there's an important table to mention , that is the *sys_call_table*
The table contains function pointers to the kernel level system calls handlers.

```
fnptr_t sys_call_table[] =  { sys_exec ,
                                 sys_fork ,
                              sys_printf
                       ....
                            };
```

Listing[5-3] System call table.

The first thing the call sets is the index for the correct function pointer in that table , indexes are defined in
include/sys.h

```
     asm("movl %0,%%eax" :: "a"(S_PRINTF));   ;set call index
     asm("movl %0,%%ebx":: "a"(1));          ;for the function
     asm("movl %0,%%ecx": :"=g"(str));   ;function argument
     asm("int %0" : :"a"(0x30));         ;SYSCALL interrupt
```

Listing[5-4] printf.c prototype

after the function issues the interrupt ,and the interrupt handler finds that it's a SYSCALL interrupt , it'll call map_syscall function
and pass the current cpu_stat structre to it , the function will index the sys_call_table and call the function with the arguments in ecx register
and return the return value (error code) of the call.

```
s_errno= (sys_call_table [cpu_state->eax])((char *)cpu_state->ecx);
     if(s_errno < 0 ) {return -s_errno;}  //error code
     else {return 0;}
```

Listing[5-5] mapping system calls

6. Process Management:

6.1 Loading Process:

Currently we're only support ELF formats , *elf_load_to_proc* reads the binary from disk and populates the proc structure with the process information and sets the entry point.

```
typedef struct proc {
    gpr_regs_t   gpr_regs;
    seg_regs_t   seg_regs;
    reg_t        eip;
    uint32_t     cs;
    reg_t        eflags;
    reg_t        esp;
    uint32_t     ss;
    uint32_t proc_id;
    uint32_t proc_status;
    pde_t        *page_directory;
    uint32_t cr3;
    uint32_t preempted;
    uint32_t dequeqed;
    LIST_ENTRY(proc) link;
    LIFO_ENTRY(proc) q_link;
} proc_t;
```

Listing [6-1] Process structure

The function will remind you of kenel fetching, based on the binary offset it will seek it and read the disk's block into an elfhdr
structure , which will then iterate through the headers to copy the entire file into memory , update the page table and the CR3 register
as an initialization to the user environment.

6.2 Scheduling

Scheduling is the process of organizing the context switching between processes in order to achieve multi-tasking In prototype1 we're using a simple LIFO Time sharing Round robin (Last In First Out) scheduling algorithm

For a quick recap , LIFO refers to the way items stored in a data structure are processed. The last data to be added to the structure will be the first data to be removed. LIFO mechanisms include data structures such as stackss. A LIFO structure can be illustrated with the example of a crowded elevator. When the elevator reaches its destination, the last people to get on are typically the first to get off , the same thing applies to processes , the last process added to the queue is the first process to be taken out of the queue so that another one would take it's place.

another important concet in scheduling is context switching , a "context" is a virtual address space, the executable contained in it, its data etc.
A "context switch" occurs for a variety of reasons - because a kernel function has been called, the application has been preempted, or because it had yielded its time slice.

A context switch involves storing the old state and retrieving the new state. The actual information stored and retrieved may include EIP, the general registers, the segment registers, CR3 (and the paging structures), FPU/MMX registers, SSE registers and other things. Because a context switch can involve changing a large amount data it can be the one most costly operation in an operating system.

since Resource management is not yet implemented we currently have 2 queues , a ready queue and a running queue after a quantum of time passes .
the schedule function checks the running and ready queues , last process in the running queue will be poped and replaced with the last process in the ready queue , which is to be scheduled then a context switch to this process occurs with the *switch_address_space* function.

```
schedule(void)
{
     uint32_t idx= 0;
     proc_t *proc, *pproc;
     if(!LIFO_EMPTY(&running_procs))
     {
          pproc = LIFO_POP(&running_procs, q_link);
          printk("[*] Proc running: %d\n",pproc->proc_id);
     }
     else
          printk("[*] No running procs\n");

     if(!FIFO_EMPTY(&ready_procs))
     {
          proc = FIFO_POP(&ready_procs);
          printk("[*] Ready proc: %d\n",proc->proc_id);
     }
     else
     {
          printk("[*] No ready procs found\n");
          proc = pproc;
     }

     if(!LIFO_EMPTY(&running_procs))
          FIFO_PUSH(&ready_procs, pproc);

     LIFO_PUSH(&running_procs, proc ,q_link);

     printk("[*] Scheduling to process: %d\n", proc->proc_id);

     switch_address_space(proc);
```

Listing[6-2] proc.c

switch_address_space fools the CPU in order to switch to another address space , it needs to reset the stack top to point the the new process structure , set the cpu state to the process' value the issues an iret.

```
void
switch_address_space(proc_t *proc_to_run){
      proc_to_run->seg_regs.fs = 0x23;
      proc_to_run->seg_regs.es = 0x23;
      proc_to_run->seg_regs.gs = 0x23;
      proc_to_run->seg_regs.ds = 0x23;
      write_cr3(proc_to_run->cr3);
      asm volatile("movl %0,%%esp":: "g" (proc_to_run) : "memory");
      asm volatile("popal");
      asm volatile("popl %gs\npopl %fs\npopl %es");
      asm volatile("popl %ds");
      asm volatile("iret\n5:\n");

      while(1);
}
```

Listing[6-3] Switching to process address space.

[Prototype Two]

1. Design Model

1.1 Monolithic vs µKernel

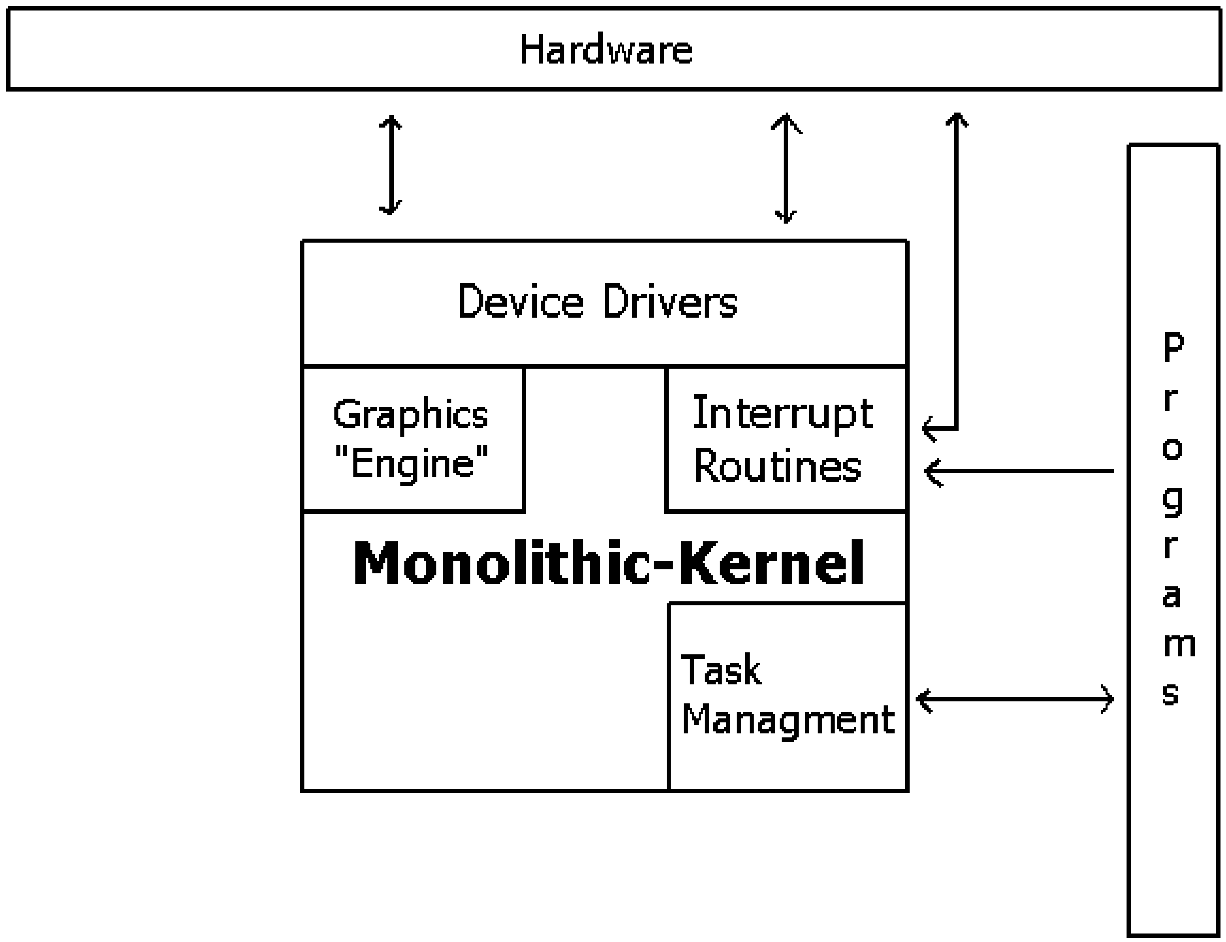
Lately in CATernel first prototype has been put to and end since we reached a crossroads point. We had make up our mind to the overall design of the kernel. We had several Options: 1- Monolithic kernel, 2- ExoKernel, 3- µKernel.

Of course the main dilemma would be between Monolithic and µKernel since most Exokernels made are for research purposes and not [rel life] daily kernels that an Operating system might use. However, a minimal Exokernel version of CATernel might be supported anyway since It won't be hard to implement.
Here, we only stand between the Monolithic design and Micro design. those two designs in particular has been a huge dilemma for years, It has began with the famous Tanenbaum - Torvalds debate. But, Let's pause here for now and start listing the main differences between the Micro and Monolithic designs.

1.1.1 Monolithic Kernel design

Normally a kernel consists of several entities mostly called "services", Like Process Management, File system management, Memory Management. A monolithic design is simple and old, but however effective. In a monolithic kernel all the kernel services are contained within one program. an example to that is the old UNIX of course, Linux and freeBSD. most of the time this (containing all services in one program) results in a huge code base which will result into hard debugging and bug tracking. and since kernel components only connect with each others in execution flow It creates a huge dependencies for kernel components that are hard to track too, at this point I recall the name Andrew Tanenbaum called the monolithic kernel ("The Big Mess"). However, Monolithic kernels have advantages that might distinguish it from a microkernel, such as:
1- It takes less code to write a monolithic kernel
2- monolithic kernels jave relatively fewer bugs, since only way to context between services is using execution flow and no external means are provided.
3- They are also relatively fast since it is only one executable entity.

In abstract form this is how monolithic kernel looks like:



Since all services of a monolithic kernel lays into the kernel executable, system calls are used to achieve privileged operations from user mode. System calls in monolithic kernels are much more than those in a Micro kernel since all privelegd services lays into kernel space. and system calls are basically interrupts, hence a code executing under a monolithic kernel will issue more interupts that the same code running under an Exokernel. However, Interrupts can hardly be an overhead since the scheduler can issues interrupts -sometimes- hundreds of times per second on normal Interactive operating systems.

Monolithic kernels have some disadvantages too, some of them crucial and some

tolerable:
1- As mentioned before, monolithic kernel contains huge relations and dependencies. Thus, a failure of a part of the kernel will result in a major failure in the whole kernel. Thus less reliability.
2- Also monolithic kernels are hard to port, since -almost- the whole kernel needs to be re-written to support a different architecture.

1.1.2 µKernels (microKernels)

are more modular than monolithic kernels, since most of the kernel services are moved to user space. Like, Networking and File systems management. And (as the name states) the real kernel is very minimal containing the Memory management and process management services for instance. The services that are runnable in user space are called servers (will show why later) and they use the real kernel to get access to hardware. a typical microKernel provides an abstraction for hardware which will make servers interactions easier and will lessen the porting pain. Surprisingly, µKernels result into more number of system calls than monolithic kernels, since all services will have to access kernel space to perform privileged operations since most of the services rely on devices.
Unlike monolithic kernels, µKernels come up with a service that monolithic kernel might not even need (not realistic kernels of course) which is IPC(Inter process communication). Inter process communication allows running processes to communicate and send and receive data. That is why kernel entities in user space are called servers, because user programs follow the client server model. they communicate with the kernel service to perform an operation and the server does that for them. µKernels however have some advantages such as:

1- µKernels are reliable, that if a service crashed it can be fixed or altered. even if it cannot be fixed the kernel would still be running.
2- Testing is easy on µKernels since loadable software doesn't require you to reload the whole kernel.
3- Easier to maintain.
4- µKernels can be easier to adapt for working under a distributed system.

µKernels disadvantages:

1- Very complicated process management.
2- IPC bugs can bring the whole system down
3- Since more μKernels are more abstract, execution time of software is relatively large.

1.1.3 Modern examples:-

Monolithic kernels years ago were limited to UNIX and BSD and some other minimal kernels. But, Linux came out of nowhere and proved the effectiveness of monolithic kernels after being considered outdated. So, the only examples I can give about modern monolithic kernels are Linux and FreeBSD.

By then μKernels were the hub of interest and most operating systems researchers and hobbyists starts considering the μ design for their kernels, Thus most modern kernels are micro. Such as, MINIX , GNU Mach/Hurd, SPARTAN. Also Windows NT can be considered micro, but It is called hybrid since it is not fully micro.

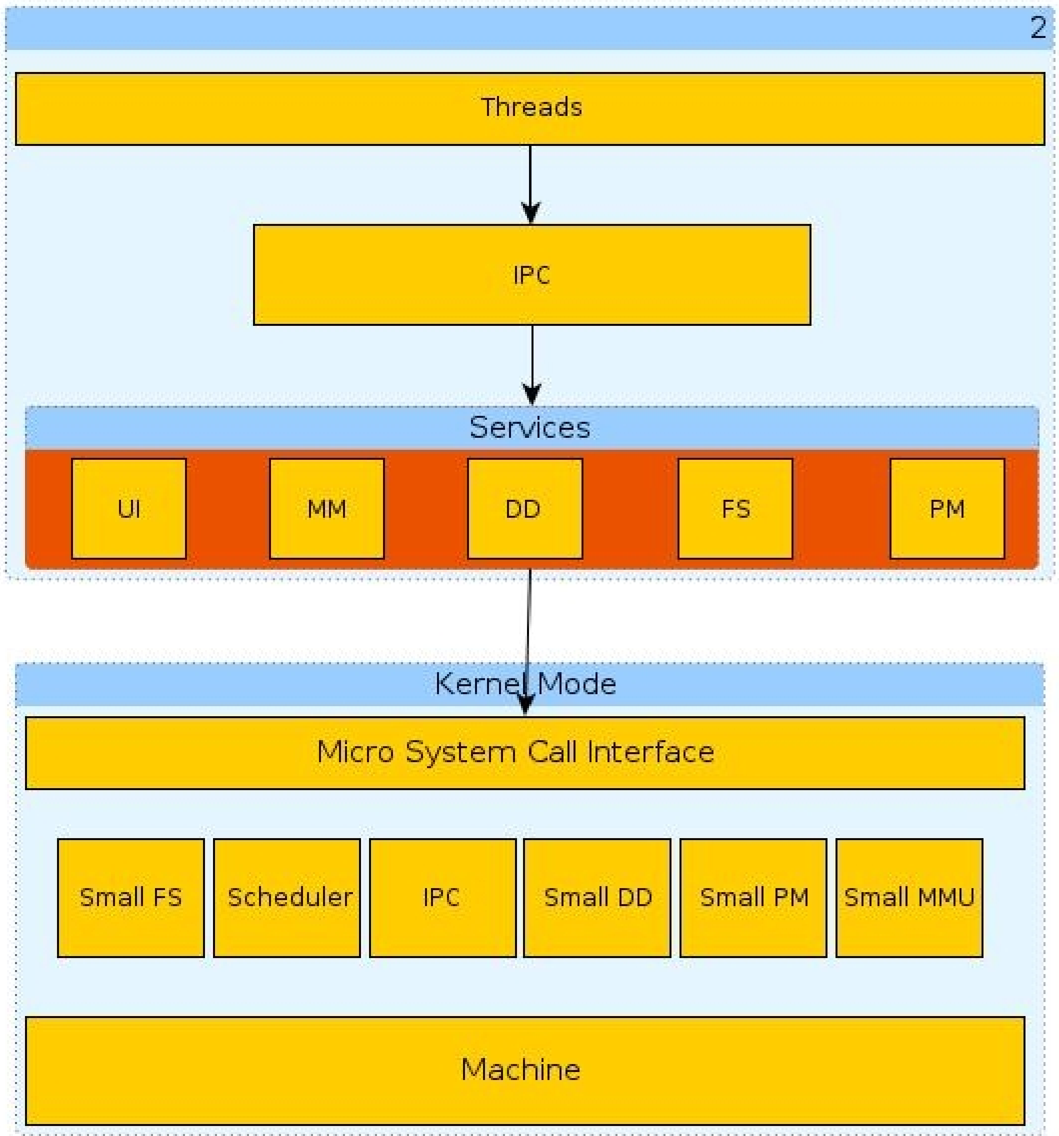
The Tanenbaum-Torvalds debate:
On the debate, Tanenbaum explicitly expressed that Linux was outdated due to the old monolithic design it uses. And he admitted that microkernels can be slower than monolithic kernels, but with some optimization they could just as fast as monolithic kernels. Also a monolithic design limits and makes portability to different machines harder (as stated above).
To that Linus did agree that micro is a better design that monolithic and portability on monolithic kernels might have issues. and stated that Linux is more functional and stable than minix.

several famous researchers joined the conversation such as:
Ken Thompson
Micheal Kufman

http://oreilly.com/catalog/opensources/book/appa.html

However, For CATernel We've favored the Micro design, Although monolithic is easier to implement. but for future issues micro kernels are easier to maintain and debug. This kernel is all about researching after all, No real work involved.

1.1.4 CATernel design model



We've established this design as our generic primary design with small resource managers at the kernel level which are enhanced by more detailed and specific manager on user level to achieve micro design of CATernel. With Small file system manager, Scheduler, IPC, Small device drivers manager, small process manager and small memory management unit on kernel mode which are interfacted by few system calls. With the more specific and dedicated managers established as services on user mode. Threads can only communicate with those servers using IPC unlike monolithic convention which uses system calls.

2. Supporting SMP

Since the overall kernel design is determined, we need to work more on basic services such as multitasking and memory management. Symmetric Multi-Processor Management, at the moment we are only able to multi-task over one processor. Of course, we will be using Intel facilities to be able to parallelize over multiple processes. There's challenges faced when it comes to parallelizing execution over a centralized Data Bus and memory, Such as:
1- Memory Coherency and (making sure that only one processor get acces to a specific memory address at a time).
2- Cache consistency, Since some caches are shared (e.g. L2/L3) corrupted data should be avoided by a processor after being written by another
3- Distribute interrupt handling amongst processors.
4- Avoiding randomized memory accesses.

2.1 Memory Coherency

Memory coherency can be simply put as follows. It is preventing a cpu from accessing a memory thunk aligned to (8/16/32/64) that is being used by another cpu(former values differ on different architecture). once a value is locked the data bus is automatically locked, so locking here is basically spin-locks. ia32 supports memory locking using atomic operations via several ways :-

2.2 Automated locking

Locking a memory address is automatically carried out on several situations
1- executing xchg instruction that reference memory
2- Updating segment descriptor
3- Updating page directory
4- Acknowledging interrupts

the last 3 cases do not really interest us, but the first instruction can be used to make a software locking to memory address. such a locking mechanism force following memory operations to be atomic. An example for that is the classic unix and very early Linux. a very basic mutual exclusion was implemented by merely exchanging the to be locked variable with a new value. Such examples are obsolete and old. With Multiprocessors architectures showing up another problem arises.

2.3 Memory ordering

For the sake of performance optimization, more modern processes adopted a new memory ordering(order of loads and stores issued) models. Instead of strong ordering which executes loads and stores with the order they are in the executable code, Modern processors use different memory models where loads and stores don't need to neccessarily done in order they're in the executable.

Due to that fact, busy waiting on a lock might go wrong. since loop is the first in the executable then modifying the lock comes second we suppose we're working on strong ordering modeled memory access. However, this can go badly wrong by obtaining the lock before executing the loop for instance, Maybe even skipping obtaining lock and starting executing critical region. Thus, cpu needs to be instructed to execute a thunk of code's memory loads and stores in the order they're in the executable. And luckily this is what 'mfence' instruction does. Memory ordering is a strict issue since it all gets messy with processors evelution. 486 and Pentium for instance had a less strict memory ordering model than strong ordering but still most of the time it would use strong ordering. P6 family however almost dropped the strong ordering model which Intel name "write ordered with store buffer forwarding". a premature implementation for the spinlocks would look like this at this point

```
lock:
        popl %eax
        movl (%eax), %ebx
        test %ebx,%ebx
        jnz lock
        incl %ebx
        xchgl %ebx,(%eax)
        mfence
        ret

unlock:
        popl %eax
        movl $0, %ebx
        xchgl %ebx,(%eax)
        mfence
        ret
```

Listing 2.1: premature spinlock

At this point Intel manuals carry on from memory instructions ordering to the relation between memory ordering and string operations and its atomic property under multiple processors and interrupts, But this is no use to us at the moment since address spaces forbid interrupts to use user stack. Moreover, generic instructions serializing is a necessity. An example mentioned in Intel manual vol.3 shows how critical instruction serialization could be crucial at some points, This example says: suppose we're switching to protected mode under a multiprocessor environment. we need all real mode instructions to be executed before protected mode is triggered. This is where serializing instructions comes in which makes sure that all instructions before it are executed before the serializing instruction is executed (e.g. CPUID)

Multi-Processor Initialization

Before mentioning the way Multiprocessing is initialized over an intel machine, we need to support and point out cpuid instruction and collecting information about the machine since we'd be having different initialization models for every machine.

Anyway, MP initialization is done by a MP protocol called Multiprocessor Specification Version. It supports controlled booting of multiple processors, It allows all IA-32 processors to be booted in the same manner and it can boot a system without defined boot processor. This MP protocol defines two types of processors which are bootstrap processor and application processors. after power one processor is selected as bootstrap processor and performs the booting up routines. Other processors are set to application processors.

Bootstrap processor stars by setting up the global data structures and initializes other application processors, following that is the Operating system code execution. when machine is powered up, Application processors wait for an IPI signal from the bootstrap processor and then performs APIC initialization procedure and put itself in halt state.

Before initializing processors, cpu information should be collected in order to perform a right initialization sequence. such information collection should be carried out after memory initialization on the BSP since values like global cpu use memory component to allocate space for itself. however we can still store it in absolute address. and we might need that in parallelizing kernel boot up.

CPU information is found in the floating point structure which is usually stored in one of Four places:
(1) in the first kilobyte of the extended BIOS data area (EBDA)
(2) the last kilobyte of base memory,
(3) the top of physical memory
(4) the BIOS read-only memory space between 0xe0000 and 0xfffff.
we'll need search these areas for the "_MP_" signature which denotes the start of the floating pointer structure.
Absence of this structure indicates that the system is not MP compliant ; if not the system will continue with UP setup

```
uint32_t length[2] = { 1024, 64 * 1024 };
        addr[0] = (uint8_t *) PA2KA(ebda ? (uint32_t)ebda : 639 * 1024);
        for (i = 0; i < 2; i++) {
                for (j = 0; j < length[i]; j += 16) {
                        if ((*((uint32_t *) &addr[i][j]) ==
                           FS_SIGNATURE) && (fps_check(&addr[i][j])==0)) {
                                fs = (fpstruct_t *) &addr[i][j];
                                goto fs_found;
                        }
                }
        }
```

Lisiting 2.2: arch/x86/mp/smp.c - find_set_fps

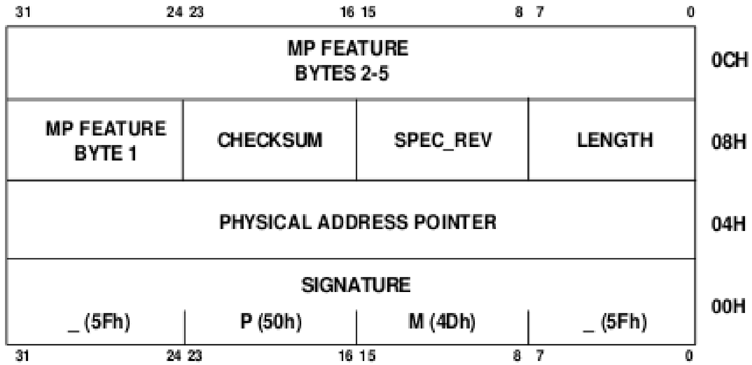


figure 2-1 floating point structure

Next step is to read the Configuration table's address form the floating point structure which will include all cpu's information
"ct=(ct_hdr *)PA2KA((uint32_t)fs->config_addr);"
The configuration table consists of a header , base table and an extended table.

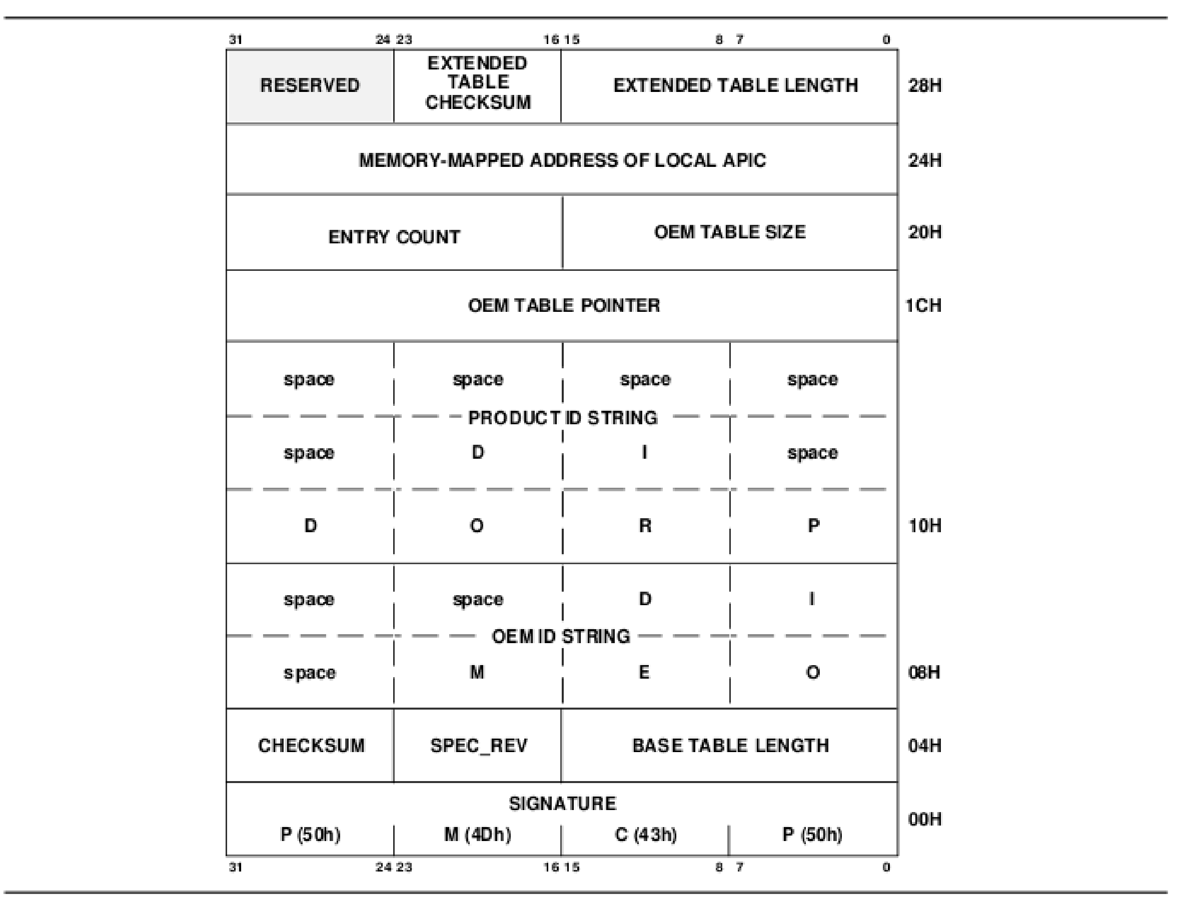


Figure 2-2 The Configuration table.

with help of the CT header -arch/x86/mp/smp.c - ct_entries() - , you'll be able to locate the processor entries and read them
most of the fields will come in handy later , refer to smp.h to see the full structure of different base entries .

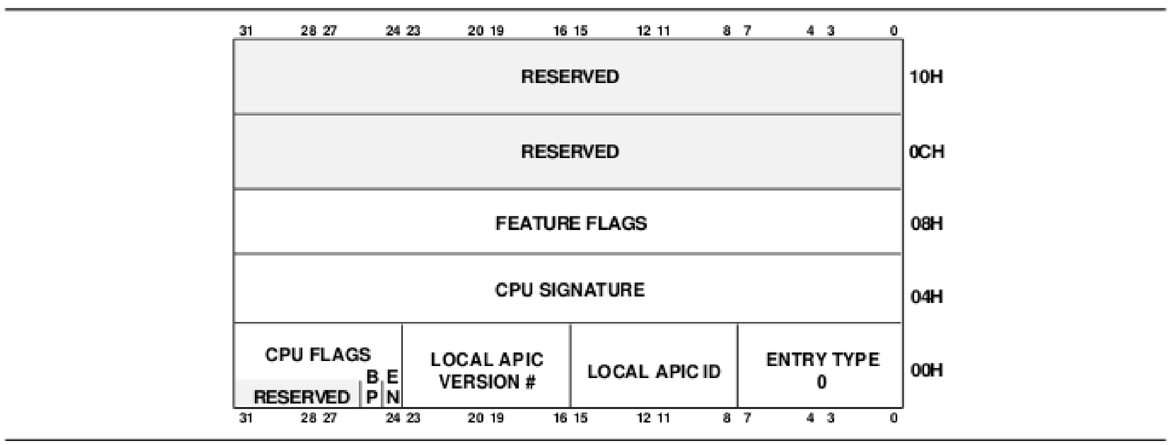


Figure 2-3 processor entry

now that we have all needed cpu info , we can finally signal them to wake up by sending an INIT IPI through the LAPIC in 0xfee00000
we're sending the IPI based on the processor's APIC ID , ofcourse it's a bad idea to send an INIT IPI or SIPI to boot strap since it's already up
so we make sure we check the bootstrap flag in the processor's entry we parsed earlier.

```
void apic_init_ipi(uint8_t lapicid)
{
	icr_t icr;
	icr.lo = lapic[ICR_LOW];
	icr.hi = lapic[ICR_HIGH];
	icr.delivery_mode = ICR_INIT;
	icr.dest_mode = PHYSICAL;
	icr.level = LEVEL_ASSERT;
	icr.trigger_mode = TRIGMOD_LEVEL;
	icr.shorthand = NO_SH;
	icr.vector = 0;
	icr.dest =lapicid;
	lapic[ICR_LOW] = icr.hi;
	lapic[ICR_HIGH] = icr.lo;
}
```

Listing 2.3: arch/x86/mp/apic.c - apic_init_ipi

Intel's specification shows that we need to wait for 20 us till an IPI is successfully delivered , then wait for
1000 us for a processor to wake up In order to send the Startup IPI which is repsonsible for making the AP jump to custom bootup code to set it up
, 200 us are needed to deliver this IPI ;

After making sure the INIT-IPI was delivered , The Application processors will wake up and wait for a S-IPI (Startup IPI) which will include the pointer to the boot up stub needed for the processors initialization , where SMPBOOT_START is the Physical address of the boot up stub shifter by 12 (e.g 000AA000 == AA) and the processor will reverse this operation to reach the stub.

```
icr.lo = lapic[ICR_LOW];
icr.vector = (SMPBOOT_START) >> 12 ;//trampoline
icr.delivery_mode = ICR_SIPI;
icr.dest_mode = PHYSICAL;
icr.level = LEVEL_ASSERT;
icr.shorthand = NO_SH;
icr.trigger_mode = TRIGMOD_LEVEL;
icr.dest=ALL_X_SELF;
lapic[ICR_LOW] = icr.lo;
     //Wait 200 us
delay(200);
apic_read_errors();
```

Listing 2.4 arch/x86/mp/apic.c - apic_s_ipi

To Sum things up , Here's A diagram for the initialization sequence.

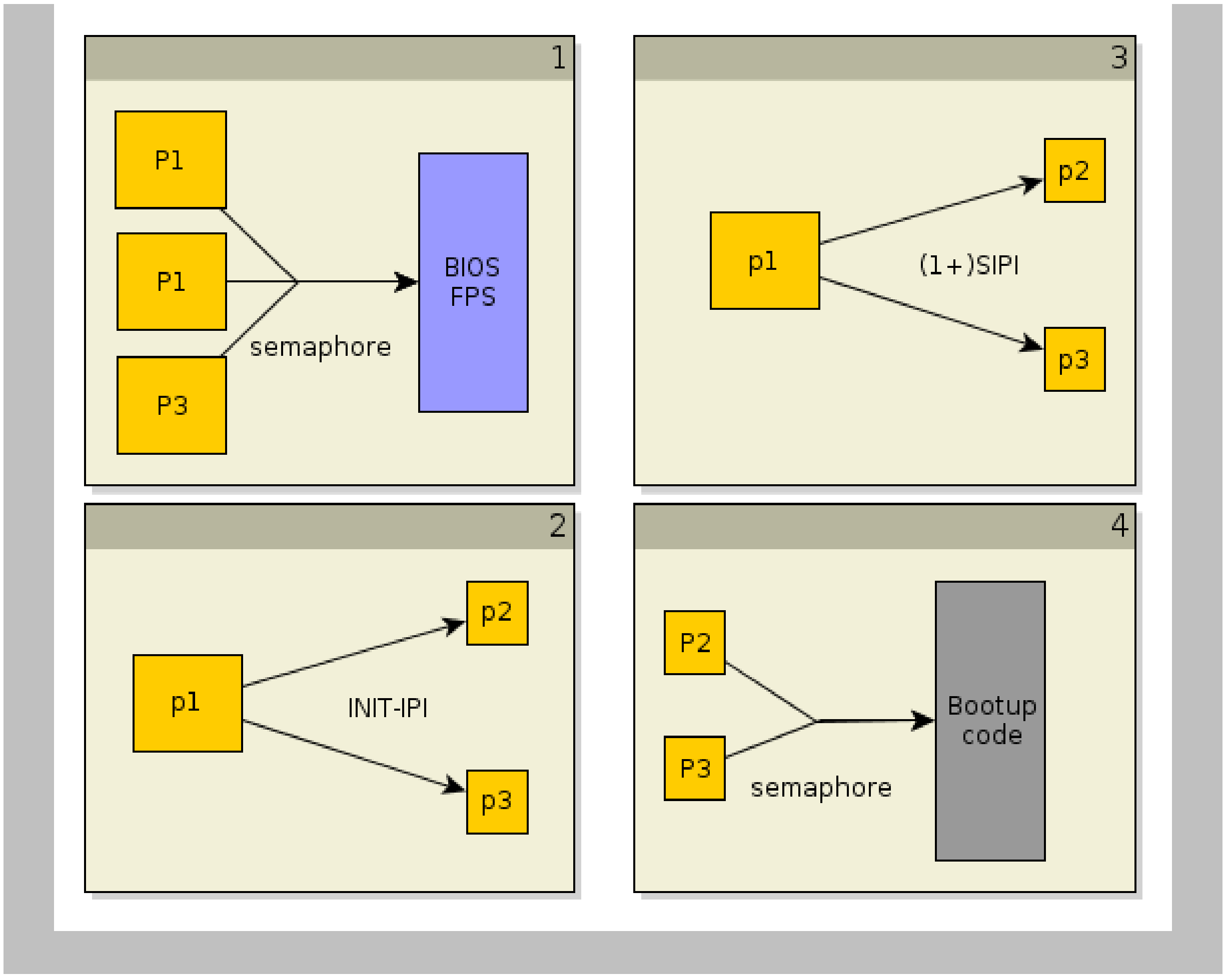


Figure 2-4 : SMP initialization sequence

3. Synchronization

After the decision to change the CATernel design into microkernel, It became inevitable to implement early synchronization primitives and IPC scheme to convert kernel components into services.

The Basic synchronization primitive is considered the spinlock mentioned in the "Supporting SMP" section. This synchronization primitive is used in active synchronization in kernel only. other type of synchronization primitives is passive synchronization, this is the most used type which includes race conditions.

In the first prototype a minimal structure of the process descriptor was implemented which was sufficient for the first prototype phase. However, moving into multiprocessing environment and to be able to provide a concurrent back-end even with a uniprocessor plus the nature of the kernel design obligated us to extend process structure. The old process descriptor structure contained :-
1- Process context
2- Process Id
3- Process Address space
4- Scheduling states
5- Pointers Proc List and running Procs LIFO

At this point also only three process states were supported
1- Process ready to run [RUNNABLE]
2- Process descriptor is undefined [EMPTY]
3- Process is not ready [NON_RUNNABLE]

Of course with such a structure processes are not able to support events, and doesn't even support them at all.

3.1 Supporting race conditions

A concurrent environment is unable to work in a consistent and proper way without race conditions of course. for a process/service to be able to process locking/unlocking events of a synchronization primitive it must either busy wait or block. However busy waiting is time wasting like in spinlocks and it probably would lead into deadlocks with prioritized processes design.
Blocking a process is simply done by two components, Scheduler and a waiting

queue. Process simply waits on race conditions locks or IO events with either a state of those states (wait interruptible – wait uninterruptible).

However There's difference between Synchronization process communication, Inter-service/kernel communication and Standard Inter-process Communication.

3.2 Synchronization Communication

Race conditions are somehow an IPC mechanism from semaphores to message queues. However there's different implementations for such synchronization mechanisms and specially using waiting queues. for example Linux kernel makes no use of waiting queues but only timers, Mach makes use of a single waiting queue and SPARTAN makes use of a multiple waiting queues with also timers same as linux only renamed to timeouts.

At this point the [Timers] implementation seems convenient. Possible ways to implement a timer are by the RTC or APIC timer, of course with having a nearly SMP environment apply an RTC timer seems inconvenient. APIC timers however also has one-shot/periodic/tsc timers. a one-shot timer might be convient also. Another way implementing timers is by setting a timer relative to Scheduler timer interrupts with the timer interrupt as a one basic unit of timer.

A pre-mature implementation for a synchronization based on timers and waiting queues is to update timers on each tick with extending implementation of timers via waiting queues and same for synchronization primitives.

3.3 Optimal Implementation

When it came to implementation various designs and schemes came up to my mind (personally) not to mention other implementations adopted by other kernels.

Semaphores

implementing the semaphore's P&V with Dijkstra's algorithm without using spinlocks isn't practical. for example to use a semaphore which only contains the integer value of the semaphore and ups and downs which directly puts an executing entity into a generic waiting process list makes it hard to determine which procs/threads are waiting for this specific resource/primitive and wake them up. a hack would be to make a list of pointers or indexes of these procs into the generic waiting list in the semaphore itself, but this would cause contention over the generic waiting list, another memory unfriendly implementation is to

make a waiting list per resource.
Within a Multiprocessor environment old irq_enable/disable won't work to keep the an up/down operation atomic. thus a spinlock implementation is a must. of course disabling all processors irqs is nonsense.

3.4 Case Studies

GNU Mach
GNU Mach impelements wait queues implicitly into a thread structure, with only one type of waiting [THREAD_WAIT]. The thread state is changed into wait so the scheduler won't pick it the next time it is invoked. However such implementation is minimal compared to the Linux implementation.

Minix
Minix implementation to IPC is rather complicated, actually very complicated. it doesn't make use of any processes blocking or suspention and ipc is done via a seperate server not as a global entity. This implementation is similar to the System V semaphores implementation. This implementation is for generic IPC purposes and Svr5 Standardization

Linux - POSIX
The way Linux implements locking and IPC waiting is by spinlocking using a unified locking way called ipc_lock. so the lock is being polled on, scheduler however makes use of this waiting into blocking a thread implicitly using the Linux waiting queues in the form of semaphore queues. However This implementation at ipc/sem.c is a System V semaphore which we don't really need at the moment. since it's for IPC Purposes not synchronization.
another implementation of the semaphore is used in kernel/semaphore explicitly makes use of waiting and task state.

SPARTAN
the SPARTAN waiting queues are very similar if not identical to the linux waiting queues, however its semaphores makes use of waiting queue explicitly. Which makes the SPARTAN semaphores and ipc more effective and fast than any other.

4. Time Management

1-Delay function
The delay function was implemented to make the cpu waste cycles for a few microseconds without being interrupted , this approach came in handy in booting up the Application processors "refer to Supporting SMP"

The delay function is hardware based , It uses PIT (i8254_calibrate_delay_loop) to calibrate the cpu to figure out how many loops are executed in 1us, which is what we will refer to as the "delay_loop_const"

The delay loop is pointless , it's only a waste of cpu cycles

```
{
	asm volatile ("movl %0 ,%%ecx\n\t"
			"0:lahf\n\t"
			"dec %%ecx\n\t"
			"jnz 0b" : : "a"(t));
}
```

Listing 4.1: asm_delay_loop

so now we can simply make the cpu wait for X microseconds simply by multiplying the time we need and the delay_loop_const .

```
void delay(uint32_t microsec)
{
	cli();
	asm_delay_loop(microsec * delay_loop_const);
	sti();
}
```

Listing 4.2: delay()

5. Virtual File System

A virtual file system is not a disk or a network file system , It's rather and abstraction that the operating system provides in order to unify differnt file system calls , For example rather that calling ext2_open you can use the virtual file system read call instead thus allowing any calls to be file system independent

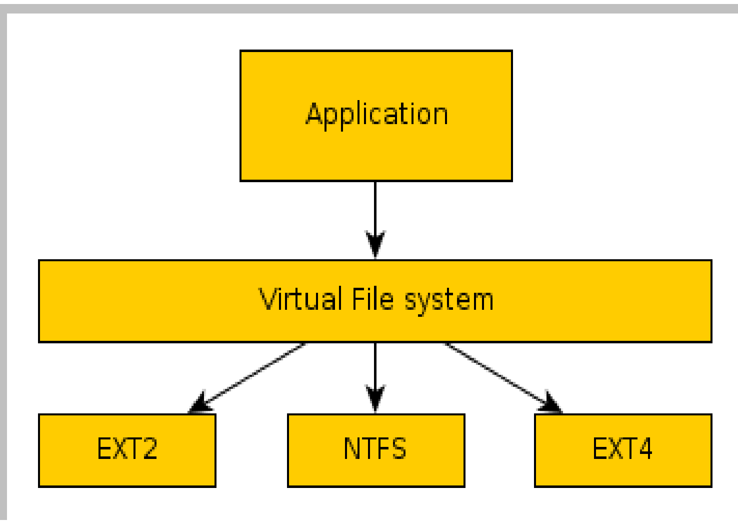


Figure 5-1 : The virtual file system.

A simple way to accomplish this is to have the node structure store specific file system function pointers which are then called by the kernel ,currently we're supporting :

Open : Called when a node is opened .
Close : Called when the node is closed.
Read : Called on read.
Write :Called on write.

Readdir : If the current node is a directory, it reads it's contents and return the next directory entry inside it.
Finddir : We also need a way of finding a child node in a directory.

5.1 Initrd

Initrd , standing for initial RAM disk is a small file system that gets loaded into memory when the kernel boots up .
Many modern operating system kernels, such as Mach , do not by themselves contain enough mechanism to get the system fully operational: they require the presence of additional software modules at boot time in order to access devices, mount file systems, etc. While these additional modules could be embedded in the main OS image along with the kernel itself, and the resulting image be split apart manually by the operating system when it receives control, it is often more flexible, more space-efficient, and more convenient to the operating system and user if the boot loader can load these additional modules independently in the first place. Our version of initrd file system is the simplest it could be as we're using a flat file system with the following format:

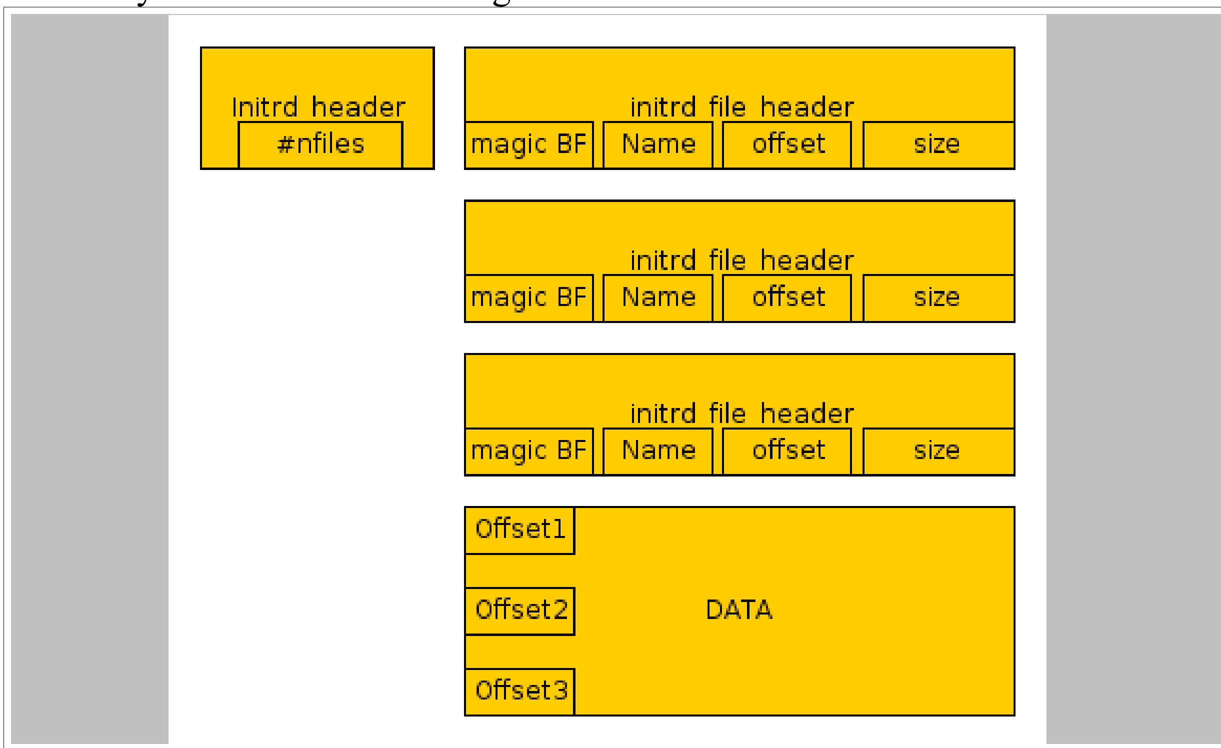


Figure 5-2 : Initrd file system

to make things easier we've included a tool that creates the initrd image in the tools/ directory. The initrd module Implements the VFS standards to match our small file system , we'll only need Read , Finddir and Readdir.
Grub loads the initrd image right after the kernel in memory ; GRUB communicates the location of this file to us via the multiboot information structure defined in multiboot.h (information about the structure fount in vfs/multiboot.info).

for more information multi boot specification refer to
http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

After loading the modules into memory via grub , It's now just a matter of applying files systems operations in order to read the contents

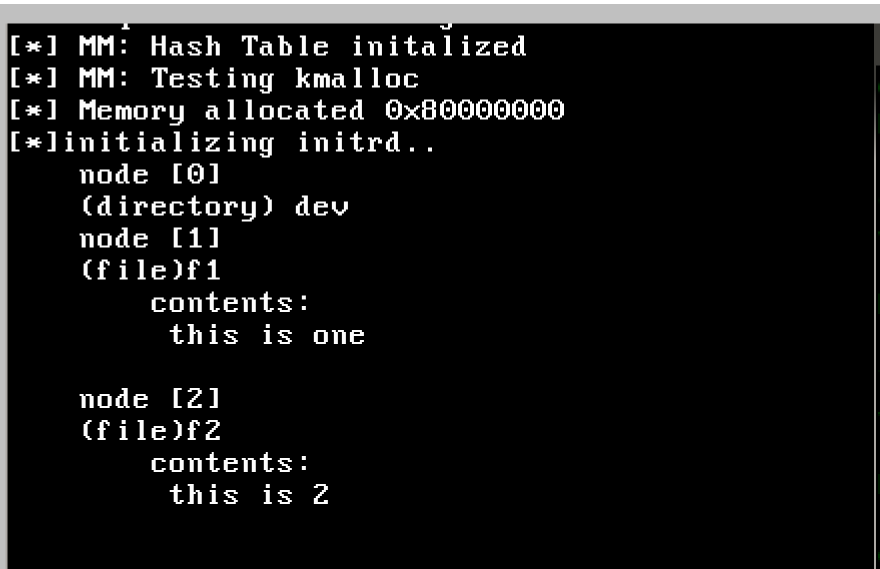


Figure 5-3 : Initrd.

After having an operational Initrd , Next step would be loading file system modules and reading proc0 from disk to move on to user space.

Appendix A – Problems faced :

On User environment initialization:-

Mainly we initialize a user environment at this point to test how effective interrupts are when issued from a user space. the way we did a user space is to simply put another elf image on the disk shifted by several sectors from kernel and load it into memory and jump to it just like we did on boot sector. a far jump with user code segment. But we faced several problems.

1-We forgot to put a writable permission on the page to be able to load data from disk to memory. But such a behavior is not acceptable. since a code segment shouldn't be writable.
2- We forgot to provide a Ring 3 (SEG|3) Or to a user segment
3- We did depend on a statically compiled/linked elf as user environmet.

Goals are:

1- to be able to load code into memory without writable permission, of course this can be done by disabling writing to page after loading the code, but this should be the role of LDT later.
2- to be able to switch to user mode. which can be done only by "Fooling the x86 cpu"
3- Provide a data structure with processes operating.

On scheduling

to schedule we need a kernel clock to issue interrupts. scheduling can be based on two criteria (time slicing, cycle slicing). however we chose to support time slicing scheduling first by using RTC(which is very naive but we chose to support legacy first). and we faced a problem that the RTC interrupt doesn't occur on protected mode. unlike the Intel 8253 PIT interrupt which does happen on protected mode. However, Some RTC timer code may not work on some real machines. The observed problem is a timer tick happened about once every second. I'm not sure why this is, and am trying to find a solution. This Makes RTC is not a serious solution for real life cases. However, on Bochs emulator an RTC interrupt occur from the slave PIC, But sadly Slave PIC doesn't support auto EOI like the master PIC therefore a handler has to issue an EOI once interrupt is handled which we did. But there was never a second interrupt from Slave PIC.

APIC Initialization

1- Due to the fact that APIC registers are at physical address 0xfee00000 we had to identity map the APIC register 4KB page, so we had to modify the kernel address space test to exclude this page from test.
2- Bochs APIC0 error, During dumping information of APIC specifically LAVR register, bochs had an error with APIC in the code. however I don't know yet if it is solvable. so this to be edited soon.
3- Different DFR Values, While dumping the values of DFR bochs it showed 0xffffffff while on qemu 0xf0000000 this might be due to the virtualization of the real APIC on hosting for the qemu. yet to be investigated too.

Context Switching & Scheduling

A proper way to make a process switch in a healthy manner from Kernel address space to its address space is to Sharing its PEB which holds information about it like page directory and cpu status. a previous approach was to share Whole process table as readable for user processes address spaces. However, this approach was poor and insecure.

Despite that a process switches to ring 0 privilege during scheduling process the scheduling structures would be still invisibile to it and would cause a page fault execption. to provide a full kernel mode an address space is being changed to kernel's to prevent any either GP or PF exceptions.

Initial ramdisk & GRUB

an initrd was an entity that we had to provide before any microkernel execlusive structures to loosen and ease the process of developing services and driver/kmods. Grub by default loads any module after its last loaded module which was in initrd case the kernel itself. This contradicted with the fact that we use a boot time allocator to allocate boot time heaps to hold early structures like Page tables and process tables. we did an easy approach in which we had to statically allocate two pages for initrd before allocating any early heaps.

CATernel

CATernel Code Documentation

2011-2013

Contents

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Module Documentation

4.1 Interrupts-and-Syscalls

Files

- file interrupt.c

    Interrupt and fault handling.
- file s_print.c

    The actual printf procedure -to be implemented- .
- file syscalls.c

    maps calls into their respective handlers.

Interrupts Initialization

- gatedesc idt [64]

    Declare IDT with 255 width interrupts.
- char ∗ x86_exception_names [ ]

    Fault names.
- int_generic
- void trap (void)
- void interrupt_printinfo (void)
- void idt_init (void)

    Initializing Interrupt Descriptor table.
- void register_exception (uint32_t index, char ∗name, uint16_t present, void(handler)(void))

    register an interrupt handling procedure
- void interrupt_init (void)
- void map_exception (uint32_t int_index, cpu_state_t ∗cpu_state)

    maps an exception/interrupt to it's handler
- uint32_t page_fault_handler (cpu_state_t ∗cpu_state)

    Page Fault Handler.

Print procedure .

- int32_t sys_printf (char ∗str)

  The actual print procedure requested by the printf system call.

System calls handler .

- fnptr_t sys_call_table [ ] = { sys_printf }

  Declare system calls table with pointers to calls handlers.
- int32_t n_calls = sizeof(sys_call_table)/sizeof(int32_t)

  keeps the size of the table for boundary checking.
- int32_t s_errno = 0

  Keeps error codes to return to the caller.
- int32_t map_syscall (cpu_state_t ∗cpu_state)

  maps the system call by offseting the sys_call_table structure using the offset in eax register.

4.1.1 Function Documentation

4.1.1.1 void idt_init (void)

Initializing Interrupt Descriptor table.

the idt init fills the idt table with generic interrupt vector which actually does nothing. later on interrupt initialization we'll set specific interrupt numbers to specific vectors [vector_-X] is independent proc, that does rely on interrupt number, which is a drawback indeed but it's fixable later on.

Definition at line 82 of file interrupt.c.

4.1.1.2 void interrupt_init (void)

Definition at line 191 of file interrupt.c.

4.1.1.3 void interrupt_printinfo (void)

Definition at line 59 of file interrupt.c.

4.1.1.4 void map_exception (uint32_t *int_index*, cpu_state_t ∗ *cpu_state*)

maps an exception/interrupt to it's handler

This function is called by the vector_x procedure and used to map faults to handler function like page fault handler and system call handler and zero division handler..etc.

Definition at line 204 of file interrupt.c.

4.1.1.5 int32_t map_syscall (cpu_state_t * cpu_state)

maps the system call by offseting the sys_call_table structure using the offset in eax register.

Parameters

cpu_state	: pointer to cpu_state_t structure

Returns

exit status / error code.

Definition at line 38 of file syscalls.c.

4.1.1.6 uint32_t page_fault_handler (cpu_state_t * cpu_state)

Page Fault Handler.

Page fault handler

Check if proc is accessing kernel space and not kernel mode.

If from kernel, The kernel panics. if from user space, stack is extended if needed or proc is killed.

Definition at line 262 of file interrupt.c.

4.1.1.7 void register_exception (uint32_t index, char * name, uint16_t present, void(handler)(void))

register an interrupt handling procedure

Parameters

index	of the interrupt OR interrupt offset in IDT
**name*	the name to be attached to interrupt.
either	interrupt is present or not.
interrupt	handler.

Definition at line 185 of file interrupt.c.

4.1.1.8 int32_t sys_printf (char * str)

The actual print procedure requested by the printf system call.

Parameters

str	: format string to print

Returns

exit status.

Definition at line 21 of file s_print.c.

4.1.1.9 void trap (void)

Definition at line 54 of file interrupt.c.

4.1.2 Variable Documentation

4.1.2.1 gatedesc idt

Declare IDT with 255 width interrupts.

Definition at line 24 of file interrupt.c.

4.1.2.2 int_generic

4.1.2.3 int32_t n_calls = sizeof(sys_call_table)/sizeof(int32_t)

keeps the size of the table for boundary checking.

Definition at line 27 of file syscalls.c.

4.1.2.4 int32_t s_errno = 0

Keeps error codes to return to the caller.

Definition at line 31 of file syscalls.c.

4.1.2.5 fnptr_t sys_call_table[] = { sys_printf }

Declare system calls table with pointers to calls handlers.

Definition at line 23 of file syscalls.c.

4.1.2.6 char* x86_exception_names[]

Initial value:

```
{
        "Divide Error #DE",
        "Debug",
        "NMI",
        "Breakpoint #BP",
        "Overflow #OV",
        "Bound Range Exceeded #BR",
```

```
"Undefined Opcode",
"Device not avalible",
"Double fault",
"Coprocessor segment overrun",
"Invalid TSS",
"Segment not present",
"Stack Segment Fault",
"General Protection",
"Page Fault",
"\0",
"x87 FPU error",
"Alignment Mask",
"Machine Check",
"SIMD exception"
}
```

Fault names.

Definition at line 29 of file interrupt.c.

4.2 Memory-Management

Files

- file init_mem.c

  Virtual Memory initialization.
- file page.c

  Intel Paging.

Virtual Memory Initialization.

Memory size is read and segmentation is reset. and kernel page directory is initialized and mapped. also other segments are initialized and mapped.

- uint32_t max_addr
- static uint32_t mem_base
- static uint32_t ext_base
- char * next_free = 0
- static uint32_t alloc_lock = 0
- struct Page * pages
- proc_t * proc_table
- struct Segdesc catgdt []

  New global descriptor table.
- struct Gdtdesc gdtdesc
- Idtdesc idtdesc

  Interrupt descriptor table descriptor.
- tss_t tss
- static uint32_t x86_read_mem_size (int x)

  Reading memory size using the CMOS RAM/RTC device.
- void memory_printinfo (void)

  Reads memory size and determine the pages count needed to map it.
- void scan_memory (void)
- void * allocate (uint32_t n, uint32_t align)

  a boot time allocation function, allocate heaps.
- void init_tss (void)

  sets up the Task state segment and task register.
- void x86_setup_memory (void)

  main memory initialization function.

Paging.

Programming the hardware support, Initially 32-bit paging is used. and Segment register refer to the whole memory since we'll be operating on protected paging memory mode. and Kernel is mapped to Virtual address 0xF0000000. Paging is set recursively by making cr3 refer to the paging directory.

- uint32_t page_count
- pde_t ∗ global_pgdir
- uint32_t global_cr3
- char ∗ next_free
- struct Page ∗ pages
- static struct PageList free_pages
- void x86_paging_init (void)

  Initiate global pages.
- void x86_page_init (struct Page ∗page)

  Initiate a page entry.
- int x86_page_alloc (struct Page ∗∗page_byref)

  Allocate a page page is removed from the free pages list and initiated.
- void x86_page_free (struct Page ∗page)

  given page is freed.
- void x86_page_detach (struct Page ∗page)

  when [un]refering a page, ref member is decremented.
- pte_t ∗ x86_pgdir_find (pde_t ∗pgdir, const void ∗va, int allocate)

  find a page, if doesn't exist allocate it
- struct Page ∗ x86_page_lookup (pde_t ∗pgdir, void ∗va, pte_t ∗∗pte)

  looks up a page refers to vaddr.
- void x86_page_remove (pde_t ∗pgdir, void ∗va)

  removes a page that refers to a given Virtual address from a page directory.
- int x86_page_insert (pde_t ∗pgdir, struct Page ∗page, void ∗va, uint32_t perm)

  inserts a page that refers to a given va, into a given page directory.
- void map_segment_page (pde_t ∗pgdir, vaddr_t linear, size_t size, paddr_t physical, int perm)

  maps a segment to a virtual address on a specific page directory.
- void x86_test_pgdir (void)

4.2.1 Function Documentation

4.2.1.1 void∗ allocate (uint32_t *n*, uint32_t *align*)

a boot time allocation function, allocate heaps.

kernel code has an 'end' symbol inserted into its code, it's externed and allocation starts from the end of kernel code.

Definition at line 150 of file init_mem.c.

4.2.1.2 void init_tss (void)

sets up the Task state segment and task register.

sets the TSS according to the base envirnoment to carry out successful interrupt handling from user space.

Definition at line 177 of file init_mem.c.

4.2.1.3 void map_segment_page (pde_t * *pgdir*, vaddr_t *linear*, size_t *size*, paddr_t *physical*, int *perm*)

maps a segment to a virtual address on a specific page directory.

Parameters

*pde_t**	page directory to use.
vaddr_t	virtual address to use.
size_t	segment size.
paddr_t	physical address of segment start.
int	page permissions.

Definition at line 303 of file page.c.

4.2.1.4 void memory_printinfo (void)

Reads memory size and determine the pages count needed to map it.

Definition at line 107 of file init_mem.c.

4.2.1.5 void scan_memory (void)

Definition at line 121 of file init_mem.c.

4.2.1.6 int x86_page_alloc (struct Page ** *page_byref*)

Allocate a page page is removed from the free pages list and initiated.

Parameters

struct	Page** reference to page pointer.

Returns

0 if success, -1 if fails

Definition at line 99 of file page.c.

4.2.1.7 void x86_page_detach (struct Page * *page*)

when [un]refering a page, ref member is decremented.

Parameters

struct	Page* page to detach from calling execusion stream.

Definition at line 135 of file page.c.

4.2.1.8 void x86_page_free (struct Page * *page*)

given page is freed.

Parameters

struct	Page*, the page to free

A page is freed by inserting it into the free pages list.

Definition at line 122 of file page.c.

4.2.1.9 void x86_page_init (struct Page * *page*)

Initiate a page entry.

Parameters

struct	Page*, reference to page that to be initialized.

Page memory is set to Zeros.

Definition at line 86 of file page.c.

4.2.1.10 int x86_page_insert (pde_t * *pgdir,* struct Page * *page,* void * *va,* uint32_t *perm*)

inserts a page that refers to a given va, into a given page directory.

Parameters

*pde_t**	page directory to insert page into.
struct	Page*, the page to insert.
*void**	Virtual address to use.
uint32_t	permissions on a page.

Returns

0 if success, -1 if fail.

a previously set page is inserted into a page directory and refer to a given virtual address no matter what physical address is, and permission flags are set on a page.

Definition at line 248 of file page.c.

4.2.1.11 struct Page∗ x86_page_lookup (pde_t ∗ *pgdir,* void ∗ *va,* pte_t ∗∗ *pte*) [read]

looks up a page refers to vaddr.

Parameters

pde_t	page directory to search.
void∗	virtual address of desired page.
pte_t∗∗	page table reference to set.

Returns

the found page.

the functions looks up for a page into a page directory and return back to values in two different manner.

it returns the desired page address if found. and the page table address is set to a given paramter.

Definition at line 197 of file page.c.

4.2.1.12 void x86_page_remove (pde_t ∗ *pgdir,* void ∗ *va*)

removes a page that refers to a given Virtual address from a page directory.

Parameters

pde_t∗	page directory to remove page from.
void∗	virtual address the page refers to.

a virtual address is looked up in a page directory and detached. then the TLB (Translate Lookaside Buffer) is updated to avoid misbehaviour or page faults.

Definition at line 221 of file page.c.

4.2.1.13 void x86_paging_init (void)

Initiate global pages.

1-initalize the global free pages, since they all free at the start. whatever that means.

2- Loop around the page list and map it to the free pages.

3- Remove I/O Hub Mapping. we need to remove the io hole from pages

Definition at line 44 of file page.c.

4.2.1.14 pte_t∗ x86_pgdir_find (pde_t ∗ *pgdir,* const void ∗ *va,* int *allocate*)

find a page, if doesn't exist allocate it

Parameters

pde_t	Page directory to search into.
const	void∗ virtual address to allocate pages for.
int	allocation flag, if set allocate a page.

Returns

pte_t∗ page table containing the created or existing page.

the function searches for the page table containing the page that refers to a given virtual address, if it doesn't exist and allocate flag is unset, NULL is returned. If it doesn't exist and allocate flag is set, page is created and it's parent page table is returned.

Definition at line 160 of file page.c.

4.2.1.15 static uint32_t x86_read_mem_size (int *x*) [static]

Reading memory size using the CMOS RAM/RTC device.

Parameters

int	the memory range to read, either base memory or extended.

Returns

memory size.

Definition at line 99 of file init_mem.c.

4.2.1.16 void x86_setup_memory (void)

main memory initialization function.

this function carries out paging initialization and Interrupt handling initialization. and maps kernel important segments into page directory before refetching execusion after activating CR0.PG. At this point paging is on

Definition at line 200 of file init_mem.c.

4.2.1.17 void x86_test_pgdir (void)

test the global page directory

Definition at line 322 of file page.c.

4.2.2 Variable Documentation

4.2.2.1 uint32_t alloc_lock = 0 [static]

Definition at line 39 of file init_mem.c.

4.2.2.2 struct Segdesc catgdt[ ]

Initial value:

```
{
        SEG_NULL,
        [1] = SEGMENT(0xffffffff, 0, SEGACS_RW | SEGACS_X),
        [2] = SEGMENT(0xffffffff, 0, SEGACS_RW),
        [3] = SEGMENT(0xffffffff, 0x000000, SEGACS_RW | SEGACS_USR | SEGACS_X),
        [4] = SEGMENT(0xffffffff, 0x0, SEGACS_RW | SEGACS_USR),

        [5] = SEG_NULL

}
```

New global descriptor table.

Since paging is on, Discard segment registers, and refer to full memory, with different DPL values. Segments are.

1- Kernel code segment.

2- Kernel Data segment.

3- User code segment.

4- User Data segment.

5- TSS empty segment.

Definition at line 58 of file init_mem.c.

4.2.2.3 uint32_t ext_base [static]

Definition at line 37 of file init_mem.c.

4.2.2.4 struct PageList free_pages [static]

Definition at line 31 of file page.c.

4.2.2.5 struct Gdtdesc gdtdesc

Initial value:

```
{
        sizeof(catgdt)-1,
        (unsigned long) catgdt

}
```

Definition at line 76 of file init_mem.c.

4.2.2.6 uint32_t global_cr3

Definition at line 26 of file page.c.

4.2.2.7 pde_t* global_pgdir

Definition at line 25 of file page.c.

4.2.2.8 Idtdesc idtdesc

Initial value:

```
 {
        (256*sizeof(gatedesc))-1,
        (unsigned long) idt

}
```

Interrupt descriptor table descriptor.

Definition at line 85 of file init_mem.c.

4.2.2.9 uint32_t max_addr

Definition at line 36 of file init_mem.c.

4.2.2.10 uint32_t mem_base [static]

Definition at line 37 of file init_mem.c.

4.2.2.11 char∗ next_free

Definition at line 38 of file init_mem.c.

4.2.2.12 char∗ next_free = 0

Definition at line 38 of file init_mem.c.

4.2.2.13 uint32_t page_count

This Macro returns paddr of page containing the given virtual address

#define VA2PA(pgdir, va)\ ({ \ pte_t ∗pte; \ pgdir = &pgdir[PGDIRX(va)]; \ if(!(∗pgdir & PAGE_PRESENT)) \ ((paddr_t) ∼0x0); \ pte = ((pte_t ∗) PA2KA(pgdir->address)) \ if(!(pte[PGTBLX(va)] & PAGE_PRESENT)) \ ((paddr_t) ∼0x0); \ else \ pte[PGTBLX(va)]->address; \ })

Definition at line 24 of file page.c.

4.2.2.14 struct Page∗ pages

Definition at line 29 of file page.c.

4.2.2.15 struct Page∗ pages

Definition at line 29 of file page.c.

4.2.2.16 proc_t∗ proc_table

Definition at line 27 of file proc.c.

4.2.2.17 tss_t tss

Definition at line 91 of file init_mem.c.

4.3 Multiprocessors

Files

- file apic.c

  supporting xAPIC for MPs

APIC

- uint32_t vector_addr = 0x00022000
- uint32_t ∗ lapic = (uint32_t ∗)(0xfee00000)
- void lapic_init (void)

  Soft Initiates the LAPIC.
- void soft_lapic_enable (void)

  Soft To enable the Local APIC to receive interrupts you also have to set bit 8 in the Spurious Interrupt Vector Register.
- void msr_lapic_enable (void)

  APIC setting IA32_APIC_BASE Model Specific Register (MSR)
- void apic_s_ipi (icr_t icr, uint8_t lapicid)

  Sending Startup IPI to processor speicified with lapicid.
- void apic_init_ipi (uint8_t lapicid)

  Sending INIT IPI to processor speicified with lapicid.
- void apic_read_errors (void)

  Reads error field in lapic.

4.3.1 Function Documentation

4.3.1.1 void apic_init_ipi (uint8_t *lapicid*)

Sending INIT IPI to processor speicified with lapicid.

Parameters

lapicid	the processors lapic id

Returns

none

Definition at line 160 of file apic.c.

4.3.1.2 void apic_read_errors (void)

Reads error field in lapic.

Parameters

none	

Returns

none

Definition at line 188 of file apic.c.

4.3.1.3 void apic_s_ipi (icr_t *icr,* uint8_t *lapicid*)

Sending Startup IPI to processor speicified with lapicid.

Parameters

icr	Interrupt control register structure to set the options for the interrupt
lapicid	the processors lapic id

Returns

none

Definition at line 117 of file apic.c.

4.3.1.4 void lapic_init (void)

Soft Initiates the LAPIC.

Parameters

none	

Returns

none

Definition at line 29 of file apic.c.

4.3.1.5 void msr_lapic_enable (void)

APIC setting IA32_APIC_BASE Model Specific Register (MSR)

Parameters

none	

Returns

none

Definition at line 75 of file apic.c.

4.3.1.6 void soft_lapic_enable (void)

Soft To enable the Local APIC to receive interrupts you also have to set bit 8 in the Spurious Interrupt Vector Register.

Enabling LAPIC

Parameters

none	

Returns

none

Definition at line 45 of file apic.c.

4.3.2 Variable Documentation

4.3.2.1 uint32_t* lapic = (uint32_t *)(0xfee00000)

Definition at line 21 of file apic.c.

4.3.2.2 uint32_t vector_addr = 0x00022000

Definition at line 20 of file apic.c.

4.4 MultiProcessors

Files

- file delay.c

  Simple delay functions for synchronization.
- file smp.c

  SMP Support functions.

Delay

- void asm_delay_loop (uint32_t t)

  useless loop to use while delaying
- void asm_fake_loop (uint32_t t)

  useless loop to calibrate cpu
- void delay (uint32_t microsec)

  uinterruptable delay function

Delay

- const char trampoline []
- const char trampoline_end []
- fpstruct_t ∗ fs

  floating point structure
- ct_hdr ∗ ct

  configuration table.
- static ct_proc_entry ∗ processor_entries
- ct_bus_entry ∗ bus_entries
- ct_io_apic_entry ∗ io_apic_entries
- ct_io_intr_entry ∗ io_intr_entries
- ct_loc_intr_entry ∗ loc_intr_entries
- uint32_t processors_count
- uint32_t io_apic_cnt
- uint32_t bus_entry_cnt
- uint32_t io_apic_entry_cnt
- uint32_t io_intr_entry_cnt
- uint32_t loc_intr_entry_cnt
- uint8_t fps_check (uint8_t ∗base)

  checks if floating point structure is valid , the sum of all elements must be 0
- uint8_t ct_check (void)

  checks if configuration is valid , the sum of all elements must be 0
- void fsp_print (fpstruct_t ∗fs)

  parses the floating point structure
- void ct_read_hdr (void)

parses the configuration table structure

- void print_proc_entry (ct_proc_entry ∗processor_entry)

  parses the cpu entry

- void ct_entries (void)

  parses the config table and sets the entries pointers

- void find_set_fps (void)

  Searches and Sets the global Floating pointer structure. 1. search first 1K of EBDA 2. if EBDA is undefined, search last 1K of base memory 3. search 64K starting at 0xf0000.

- uint8_t cpu_is_bootstrap (ct_proc_entry ∗processor_entry)

  checks if cpu is bootstrap

- uint8_t cpu_is_enabled (ct_proc_entry ∗processor_entry)

  checks if cpu is enabled

- void map_AP_Startup (void)

  Initializes application processors.

- void ap_init (void)

4.4.1 Function Documentation

4.4.1.1 void ap_init (void)

Definition at line 291 of file smp.c.

4.4.1.2 void asm_delay_loop (uint32_t *t*)

useless loop to use while delaying

Parameters

t	number of loops

Returns

none

Definition at line 19 of file delay.c.

4.4.1.3 void asm_fake_loop (uint32_t *t*)

useless loop to calibrate cpu

Parameters

number	of loops

Returns

none

Definition at line 32 of file delay.c.

4.4.1.4 uint8_t cpu_is_bootstrap (ct_proc_entry * *processor_entry*)

checks if cpu is bootstrap

Parameters

processory-_entry	pointer to processor structure in config table

Returns

1 if bootstrap 0 if not

Definition at line 261 of file smp.c.

4.4.1.5 uint8_t cpu_is_enabled (ct_proc_entry * *processor_entry*)

checks if cpu is enabled

Parameters

processory-_entry	pointer to processor structure in config table

Returns

1 if enabled 0 if not

Definition at line 273 of file smp.c.

4.4.1.6 uint8_t ct_check (void)

checks if configuration is valid , the sum of all elements must be 0

Parameters

base	base of table

Returns

0 if true 1 if false

Definition at line 67 of file smp.c.

4.4.1.7 void ct_entries (void)

parses the config table and sets the entries pointers

Parameters

none	

Returns

none

Definition at line 147 of file smp.c.

4.4.1.8 void ct_read_hdr (void)

parses the configuration table structure

Parameters

none	

Returns

none

Definition at line 110 of file smp.c.

4.4.1.9 void delay (uint32_t *microsec*)

uinterruptable delay function

Parameters

microseconds	to wait

Returns

none

Definition at line 47 of file delay.c.

4.4.1.10 void find_set_fps (void)

Searches and Sets the global Floating pointer structure. 1. search first 1K of EBDA 2. if EBDA is undefined, search last 1K of base memory 3. search 64K starting at 0xf0000.

Definition at line 231 of file smp.c.

4.4.1.11 uint8_t fps_check (uint8_t * base)

checks if floating point structure is valid , the sum of all elements must be 0

Parameters

base	base of table

Returns

0 if true 1 if false

Definition at line 50 of file smp.c.

4.4.1.12 void fsp_print (fpstruct_t * fs)

parses the floating point structure

Parameters

fs	pointer to structure

Returns

none

Definition at line 96 of file smp.c.

4.4.1.13 void map_AP_Startup (void)

Initializes application processors.

Parameters

void	

Returns

void

Definition at line 286 of file smp.c.

4.4.1.14 void print_proc_entry (ct_proc_entry * *processor_entry*)

parses the cpu entry

Parameters

processor_entry	pointer to ct_proc_entry structure

Returns

none

Definition at line 132 of file smp.c.

4.4.2 Variable Documentation

4.4.2.1 ct_bus_entry* bus_entries

Definition at line 32 of file smp.c.

4.4.2.2 uint32_t bus_entry_cnt

Definition at line 40 of file smp.c.

4.4.2.3 ct_hdr* ct

configuration table.

Definition at line 28 of file smp.c.

4.4.2.4 fpstruct_t* fs

floating point structure

Definition at line 24 of file smp.c.

4.4.2.5 uint32_t io_apic_cnt

Definition at line 39 of file smp.c.

4.4.2.6 ct_io_apic_entry* io_apic_entries

Definition at line 33 of file smp.c.

4.4.2.7 uint32_t io_apic_entry_cnt

Definition at line 41 of file smp.c.

4.4.2.8 ct_io_intr_entry* io_intr_entries

Definition at line 34 of file smp.c.

4.4.2.9 uint32_t io_intr_entry_cnt

Definition at line 42 of file smp.c.

4.4.2.10 ct_loc_intr_entry* loc_intr_entries

Definition at line 35 of file smp.c.

4.4.2.11 uint32_t loc_intr_entry_cnt

Definition at line 43 of file smp.c.

4.4.2.12 ct_proc_entry* processor_entries [static]

Definition at line 31 of file smp.c.

4.4.2.13 uint32_t processors_count

Definition at line 38 of file smp.c.

4.4.2.14 const char trampoline[ ]

4.4.2.15 const char trampoline_end[ ]

4.5 CPU-Initialization

Files

- file processor.c

    Identify and initialize cpu structures.

CPU-Identification

- char * cpu_vendors [3]
- cpu_t * cpu
- void processor_printinfo ()
- void processor_identify (void)

4.5.1 Function Documentation

4.5.1.1 void **processor_identify** (void)

Definition at line 23 of file processor.c.

4.5.1.2 void **processor_printinfo** ( )

Definition at line 47 of file processor.c.

4.5.2 Variable Documentation

4.5.2.1 **cpu_t* cpu**

Definition at line 19 of file processor.c.

4.5.2.2 char* **cpu_vendors[3]**

Initial value:

```
{
        "Undefined\0",
        "Intel\0",
        "AMD\0"
}
```

Definition at line 13 of file processor.c.

4.6 Drivers

Files

- file clock.c

  Clock Low level controller. sets periodic interrupts and power state.
- file cmos.c

  CMOS Driver.
- file i8254.c

  Intel 8253/8254 PIT(Programmable Interrupt Timer) Controller.
- file i8259.c

  Intel 8259 PIC(Programmable Interrupt Controller) Low Level Driver.
- file kbc.c

  Keyboard driver.
- file video.c

  Color Graphics Adapter (CGA) driver.

RTC Driver.

- void clock_enable_rtc (void)

CMOS Driver.

supports status A and B Power options , Refer to boch's Ports.lst for details.

- uint8_t cmos_set_power_stat (uint8_t stat)

  supports CMOS Status B power options.
- uint32_t cmos_get_reg (uint8_t value)

  Gets RTC registers values.
- uint32_t cmos_set_reg (uint8_t index, uint8_t value)

  set RTC registers values.

Intel 8254 PIT

- uint32_t delay_loop_const
- void i8254_init (void)

  Initiate the PIT to interrupt every time slice.
- void i8254_calibrate_delay_loop (void)

  calibrates the number of loops per 1 ms
- #define MAGIC_NUMBER 1194
- #define LOOPS 150000

i8259-PIC Driver.

- void i8259_init (void)

  Initializing the PIC to work under Protected mode.
- void i8259_disable (void)

  Disables the PIC IRQs.
- void i8259_mask_irq (uint16_t irq)
- void i8259_unmask_irq (uint16_t irq)
- uint16_t i8259_read_isr (void)

  get the in service interrupt.
- uint16_t i8259_read_irr (void)

  reads the IRR register which holds raised interrupt priority
- void i8259_eoi (uint8_t irq)

  End of interrupt instruction, used for ISRs.

PS/2 Keyboard Driver

This file contains keyboard mapping and input handling functions.

- static uint8_t shiftcode [256]

  codes to shift character values.
- static uint8_t togglecode [256]

  button codes that toggle button values
- static uint8_t normalmap [256]

  the normal keyboard map.
- static uint8_t shiftmap [256]

  the keyboard map once shiftcode is pressed.
- static uint8_t ctlmap [256]

  the keyboard map once control is pressed.
- static uint8_t ∗ charcode [4]
- int kbc_data (void)

  keyboard input controller driver
- void kbc_interrupt (void)

  calls console interrupt which is used in character read.
- #define ESCODE (1<<6)
- #define CTL (1<<1)
- #define SHIFT (1<<0)
- #define ALT (1<<2)
- #define CAPSLOCK (1<<3)
- #define NUMLOCK (1<<4)
- #define SCROLLLOCK (1<<5)
- #define NUL 0
- #define CL(x) ((x)-'@')

CGA Driver

An early initialization for the console can be useful in debugging and logging operations. here screen is initialized and get ready for I/O operations.

- static cga_attr

  holds the colors attribute of the written character
- uint16_t cga_get_pos (void)

  gets the cursor position on the screen.
- void cga_set_attr (uint16_t c)

  sets global color attribute to a given value.
- void cga_clear (void)

  clears the screen.
- void cga_set_pos (uint16_t pos)

  sets the position to a given value on the screen map
- void cga_init (void)

  initiates CGA.
- void cga_putc (int c)

  writes a character to screen using cga.
- void cga_putstr (char *c)

  uses cga_putc to write a string

4.6.1 Define Documentation

4.6.1.1 #define ALT (1<<2)

Definition at line 23 of file kbc.c.

4.6.1.2 #define CAPSLOCK (1<<3)

Definition at line 24 of file kbc.c.

4.6.1.3 #define CL(*x*) ((x)-'@')

Definition at line 97 of file kbc.c.

4.6.1.4 #define CTL (1<<1)

Definition at line 21 of file kbc.c.

4.6.1.5 #define ESCODE (1<<6)

Definition at line 20 of file kbc.c.

4.6.1.6 #define LOOPS 150000

Definition at line 17 of file i8254.c.

4.6.1.7 #define MAGIC_NUMBER 1194

Definition at line 16 of file i8254.c.

4.6.1.8 #define NUL 0

Definition at line 49 of file kbc.c.

4.6.1.9 #define NUMLOCK (1<<4)

Definition at line 25 of file kbc.c.

4.6.1.10 #define SCROLLLOCK (1<<5)

Definition at line 26 of file kbc.c.

4.6.1.11 #define SHIFT (1<<0)

Definition at line 22 of file kbc.c.

4.6.2 Function Documentation

4.6.2.1 void cga_clear (void)

clears the screen.

Definition at line 41 of file video.c.

4.6.2.2 uint16_t cga_get_pos (void)

gets the cursor position on the screen.

The get position function gets the cursor position by reading it from the data register, by indexing the cga index register.

Definition at line 28 of file video.c.

4.6.2.3 void cga_init (void)

initiates CGA.

This function sets global variables to its initial values. sets **start** to address of the C-GA buffer. and sets current character buffer to the start value. and initalizes cga color attributes to none.

Definition at line 58 of file video.c.

4.6.2.4 void cga_putc (int *c*)

writes a character to screen using cga.

Parameters

int	c the character to be written.

The CGA putc is relatively big function and similar somehow to the printing function, here we take only one character at a time and print it to screen and attribute is set to every printed char. then character type is being checked and every character has its own operation, for example:

Backspace: cursor position is decremented by one.

New-Line: a whole line character count is added to the cursor position

After a chacter is written, screen checks if it needs to be shifted up.

Definition at line 69 of file video.c.

4.6.2.5 void cga_putstr (char ∗ *c*)

uses cga_putc to write a string

Definition at line 118 of file video.c.

4.6.2.6 void cga_set_attr (uint16_t *c*)

sets global color attribute to a given value.

Parameters

uint16_t	c the color attribute

Definition at line 37 of file video.c.

4.6.2.7 void cga_set_pos (uint16_t *pos*)

sets the position to a given value on the screen map

Parameters

uint32_t	pos the position to be set.

Function behavious is similar to cga_get_pos

Definition at line 50 of file video.c.

4.6.2.8 void clock_enable_rtc (void)

Definition at line 15 of file clock.c.

4.6.2.9 uint32_t cmos_get_reg (uint8_t *value*)

Gets RTC registers values.

Parameters

uint8_t	value , the register to be read.

Returns

uint32_t val , the value of the register.

Definition at line 58 of file cmos.c.

4.6.2.10 cmos_set_power_stat (uint8_t *stat*)

supports CMOS Status B power options.

Parameters

uint8_t	stat , the value of the option to be set.

Returns

uint8_t New_Stat , the CMOS new power status.

Definition at line 26 of file cmos.c.

4.6.2.11 uint32_t cmos_set_reg (uint8_t *index,* uint8_t *value*)

set RTC registers values.

Parameters

uint8_t	value , the value to set register to.
uint8_t	index, index of the register.

Returns

uint32_t val , the value of the register.

Definition at line 83 of file cmos.c.

4.6.2.12 void i8254_calibrate_delay_loop (void)

calibrates the number of loops per 1 ms

Parameters

none	

Returns

none

Definition at line 43 of file i8254.c.

4.6.2.13 void i8254_init (void)

Initiate the PIT to interrupt every time slice.

PIT is programmed to interrupt 20 times per second which is relatively big time interval to schedule on

Definition at line 28 of file i8254.c.

4.6.2.14 void i8259_disable (void)

Disables the PIC IRQs.

This function is mostly used before setting up APIC to usage.

Definition at line 77 of file i8259.c.

4.6.2.15 void i8259_eoi (uint8_t *irq*)

End of interrupt instruction, used for ISRs.

Parameters

uint8_t	irq, the Interrupt index that has ended.

In interrupt service routines an [end of interrupt] must be issued after fullfilling the service routine. In order to avoid issuing an eoi for an interrupt raised on slave PIC by an ISR of an interrupt in service on master PIC, an eoi is issued for a single PIC at a time.

Definition at line 169 of file i8259.c.

4.6.2.16 void i8259_init (void)

Initializing the PIC to work under Protected mode.

Since PIC was designed to work under real mode, it worked under the IVT which's IRQ indexes conflict with the default Exceptions in IDT defined by Intel, so we need to offset the default IRQs with 0x20 slot or whatever it requires to avoid conflict with the Intel default Exceptions.

PIC Initialization is done by issuing four Initialization command words. ICW1 = 0x11 both ICW4 Needed and ICW1 ISSUED Flags are set. ICW2 = the IRQ Base offset. ICW3 = The PIC Slave attachment situation ICW4 = Additional info. 00010001 Flags. ICW1 ISSUED - CASCADE MODE - EDGE TRIGGERED MODE - ICW4 NEEDED

00000001 Flags. 8086/8088 MODE - NORMAL EOI - NON BUFFERED MODE - NO SPECIAL FULL NESTED MODE

Output flags to PICs

Definition at line 32 of file i8259.c.

4.6.2.17 void i8259_mask_irq (uint16_t *irq*)

Definition at line 91 of file i8259.c.

4.6.2.18 uint16_t i8259_read_irr (void)

reads the IRR register which holds raised interrupt priority

Returns

returns the Interrupt request register.

Definition at line 150 of file i8259.c.

4.6.2.19 uint16_t i8259_read_isr (void)

get the in service interrupt.

Returns

returns the In service interrupt mask

Definition at line 136 of file i8259.c.

4.6.2.20 void i8259_unmask_irq (uint16_t *irq*)

Definition at line 113 of file i8259.c.

4.6.2.21 int kbc_data (void)

keyboard input controller driver

Returns

character read from screen

The task of this function is to read the keyboard input, It makes sure that there's data in the input register, then reads the data port. and maps input data to a character depending on the code generated.

Definition at line 127 of file kbc.c.

4.6.2.22 void kbc_interrupt (void)

calls console interrupt which is used in character read.

Definition at line 161 of file kbc.c.

4.6.3 Variable Documentation

4.6.3.1 cga_attr [static]

holds the colors attribute of the written character

Definition at line 26 of file video.c.

4.6.3.2 uint8_t* charcode[4] [static]

Initial value:

```
 {
        normalmap,
        shiftmap,
        ctlmap,
        ctlmap
}
```

Definition at line 121 of file kbc.c.

4.6.3.3 uint8_t ctlmap[256] [static]

Initial value:

```
{
        NUL,    NUL,    NUL,    NUL,    NUL,    NUL,    NUL,    NUL,
        NUL,    NUL,    NUL,    NUL,    NUL,    NUL,    NUL,    NUL,
        CL('Q'),CL('W'),CL('E'),CL('R'),CL('T'),CL('Y'),CL('U'),CL('I'),
        CL('O'),CL('P'),NUL,    NUL,    '\r',   NUL,    CL('A'),CL('S'),
```

```
        CL('D'),CL('F'),CL('G'),CL('H'),CL('J'),CL('K'),CL('L'),NUL,
        NUL,    NUL,    NUL,    CL('\\'),CL('Z'),CL('X'),CL('C'),CL('V'),
        CL('B'),CL('N'),CL('M'),NUL,    NUL,    CL('/'),NUL,    NUL,
        [0x97] KEY_HOME,        [0x9C] '\n',
        [0xB5] CL('/'),         [0xC8] KEY_UP,
        [0xC9] KEY_PGUP,        [0xCB] KEY_LF,
        [0xCD] KEY_RT,          [0xCF] KEY_END,
        [0xD0] KEY_DN,          [0xD1] KEY_PGDN,
        [0xD2] KEY_INS,         [0xD3] KEY_DEL


}
```

the keyboard map once control is pressed.

Definition at line 102 of file kbc.c.

4.6.3.4 uint32_t delay_loop_const

Definition at line 19 of file i8254.c.

4.6.3.5 uint8_t normalmap[256] [static]

Initial value:

```
{
        NUL,0x1B,'1','2','3','4','5','6',
        '7','8','9','0','-','=','\b','\t',
        'q','w','e','r','t','y','u','i',
        'o','p','[',']','\n',NUL,'a','s',
        'd','f','g','h','j','k','l',';',
        '\'','`',NUL,'\\','z','x','c',
        'v','b','n','m',',','.','/',NUL,'*',
        NUL,' ',NUL,NUL,NUL,NUL,NUL,NUL,NUL,
        NUL,NUL,NUL,NUL,NUL,NUL,'7','8','9',
        '-','4','5','6','+','1','2','3','0',
        '.', NUL, NUL, NUL, NUL,
        [0x97] KEY_HOME,        [0x9C] '\n',
        [0xB5] '/',             [0xC8] KEY_UP,
        [0xC9] KEY_PGUP,        [0xCB] KEY_LF,
        [0xCD] KEY_RT,          [0xCF] KEY_END,
        [0xD0] KEY_DN,          [0xD1] KEY_PGDN,
        [0xD2] KEY_INS,         [0xD3] KEY_DEL
}
```

the normal keyboard map.

Definition at line 54 of file kbc.c.

4.6.3.6 uint8_t shiftcode[256] [static]

Initial value:

```
{
        [0x1D] CTL,
        [0x2A] SHIFT,
        [0x36] SHIFT,
        [0x38] ALT,
        [0x9D] CTL,
        [0xB8] ALT

}
```

codes to shift character values.

Definition at line 30 of file kbc.c.

4.6.3.7 uint8_t shiftmap[256] [static]

Initial value:

```
{
        NUL,0x1B,'!','@','#','$','%','^',
        '&','*','(',')','_','+','\b','\t',
        'Q','W','E','R','T','Y','U','I','O',
        'P','{','}','\n',NUL,'A','S','D','F',
        'G','H','J','K','L',':','"','~',
        NUL,'|','Z','X','C','V','B','N','M',
        '<','>','?',NUL,'*',NUL,' ',NUL,NUL,
        NUL,NUL,NUL,NUL,NUL,NUL,NUL,NUL,NUL,
        NUL,NUL,'7','8','9','-','4','5','6',
        '+','1','2','3','0','.', NUL,NUL,NUL,NUL,
        [0x97] KEY_HOME,        [0x9C] '\n',
        [0xB5] '/',             [0xC8] KEY_UP,
        [0xC9] KEY_PGUP,        [0xCB] KEY_LF,
        [0xCD] KEY_RT,          [0xCF] KEY_END,
        [0xD0] KEY_DN,          [0xD1] KEY_PGDN,
        [0xD2] KEY_INS,         [0xD3] KEY_DEL
}
```

the keyboard map once shiftcode is pressed.

Definition at line 78 of file kbc.c.

4.6.3.8 uint8_t togglecode[256] [static]

Initial value:

```
{
        [0x3A] CAPSLOCK,
        [0x45] NUMLOCK,
        [0x46] SCROLLLOCK
}
```

button codes that toggle button values

Definition at line 43 of file kbc.c.

4.7 Main

Files

- file cli.c

    CLI I/O facilities.

- file printf.c

    a printf family like functions

- file readline.c

    readline operation.

- file string.c

    String and memory operations, originally from linux.

- file work_it_out.c

    kernel device facilities initialization and set up start.

Defines

- #define BUFLEN 1024

Functions

- void play (void)

    this function holds initializations and setting up facilities

- void _panic_ (const char ∗file, int nline, const char ∗fmt,...)
- void time_print (void)

    a completely useless function that prints current time.

- uint8_t initrd (multiboot_info_t ∗mboot_ptr, uint32_t ∗initrd_location_ptr, uint32_t ∗initrd_end_ptr)
- void initrd_test (uint32_t initrd_location)
- void bootup (uint32_t mboot_ptr)
- char ∗ readline (const char ∗towrite)

    reads a string from keyboard into screen.

Variables

- static const char ∗ error_panic = NULL
- static char buf [BUFLEN]

Console

A generic wrapping for the display I/O.

- void console_init (void)

    initalizes console.
- void console_interrupt (int(*intr)(void))

    *reads a char from a input device interrupt. int (*intr)(void) interrupt handler.*
- void console_clear (void)
- int console_getc (void)

    reads a character from an input device [keyboard].
- void console_putc (int c)

    writes a character to screen wraps cga_putc function
- void putchr (int c)

    writes a character to screen.
- int getchar (void)

    recives a character from keyboard and write it to screen and buffer.

Printing

- static void putch (int c, int *count)

    prints a character at a time.
- void ksprintkn (void(*func)(int, int *), int *count, uintmax_t num, int base, int width, int padc)

    Prints numbers.
- int getint (va_list *ap, int lflag)

    gets the number of the va_list on a printk.
- int getuint (va_list *ap, int lflag)

    same as getint except for the signed nature of getint.
- int kvprintk (const char *format, void(*func)(int, int *), int *count, va_list ap)

    A BSD like printing function.
- vprintk (const char *format, va_list ap)

    passes arugments to kvprintk after tokenizing them
- int printk (const char *format,...)

    An ideal printing function.
- #define hex2ascii(x) ("0123456789ABCDEF"[x])

Strings and Memory.

Basic copying and sizing and setting operations.

- uint32_t strnlen (const char *str, uint32_t count)

    the function loops over the string character values till '\0' is reached or reached max value.

- void ∗ memcopy (void ∗dst, const void ∗src, uint32_t count)

  copies bytes from a memory address to another memory address
- void ∗ memset (void ∗ptr, int c, uint32_t count)

  sets a number of bytes starting from a given pointer to a given character.
- int strcmp (const char ∗s1, const char ∗s2)

Mainflow.

After kernel is loaded into IP, it fetches into **_start** function which makes a soft reboot by the bios, and reloads a new global descriptor table and resets segments. then just before it jumps to the **work_it_out** function it sets stack up and jumps to earlier mentioned function.

- void work_it_out (uint32_t mboot_ptr)

4.7.1 Define Documentation

4.7.1.1 #define **BUFLEN** 1024

Definition at line 18 of file readline.c.

4.7.1.2 #define **hex2ascii**(*x*) ("0123456789ABCDEF"[x])

Definition at line 14 of file printf.c.

4.7.2 Function Documentation

4.7.2.1 void **_panic_** (const char ∗ *file,* int *nline,* const char ∗ *fmt,* *...*)

Definition at line 45 of file play.c.

4.7.2.2 void **bootup** (uint32_t *mboot_ptr*)

Definition at line 132 of file play.c.

4.7.2.3 void **console_clear** (void)

Definition at line 40 of file cli.c.

4.7.2.4 int **console_getc** (void)

reads a character from an input device [keyboard].

Returns

int c Character read.

It issues a keyboard interrupt which issues a console interrupt, and wait for a character to be read.

Definition at line 44 of file cli.c.

4.7.2.5 void console_init (void)

initalizes console.

Initalizes the display device and sets cursor positions to zero.

Definition at line 20 of file cli.c.

4.7.2.6 void console_interrupt (int(∗)(void) *intr*)

reads a char from a input device interrupt. int (∗intr)(void) interrupt handler.

This function takes the input device interrupt handler as a paramter and reads a character from it into the console buffer.

Definition at line 28 of file cli.c.

4.7.2.7 void console_putc (int *c*)

writes a character to screen wraps cga_putc function

int c character to print.

Definition at line 57 of file cli.c.

4.7.2.8 int getchar (void)

recives a character from keyboard and write it to screen and buffer.

Returns

int char read from keyboard.

Definition at line 65 of file cli.c.

4.7.2.9 int getint (va_list ∗ *ap*, int *lflag*)

gets the number of the va_list on a printk.

Parameters

va_list∗	the list of parameters passed by printk
int	the long flag.

Returns

the passed number.

this function is issued everytime a number flag in the format string is found to fetch a number from the va_list passed.

Definition at line 49 of file printf.c.

4.7.2.10 int getuint (va_list ∗ *ap*, int *lflag*)

same as getint except for the signed nature of getint.

Parameters

va_list∗	the list of parameters passed by printk
int	the long flag.

Returns

the passed number.

Definition at line 66 of file printf.c.

4.7.2.11 uint8_t initrd (multiboot_info_t ∗ *mboot_ptr*, uint32_t ∗ *initrd_location_ptr*, uint32_t ∗ *initrd_end_ptr*)

Definition at line 80 of file play.c.

4.7.2.12 void initrd_test (uint32_t *initrd_location*)

Definition at line 98 of file play.c.

4.7.2.13 void ksprintkn (void(∗)(int, int ∗) *func*, int ∗ *count*, uintmax_t *num*, int *base*, int *width*, int *padc*)

Prints numbers.

Parameters

void	(∗)(int, int∗) a char printing function.
int∗	reference of the count variable, to store new count in.
uintmax_t	the number to print
int	base of the number.
int	digit width to print according to.
int	the padding character to print to pad a number.

this function converts numbers into their character representation the function recursively calls itself to print lower digits first.

Definition at line 28 of file printf.c.

4.7.2.14 int kvprintk (const char ∗ *format,* void(∗)(int, int ∗) *func,* int ∗ *count,* va_list *ap*)

A BSD like printing function.

Parameters

const	char∗ format string.
void	(∗)(int,int∗) a char printing function.
int∗	reference of count variable, to store count in.
va_list	the list of paramters passed by printk->vprintk

Returns

count of chars written

Definition at line 78 of file printf.c.

4.7.2.15 void ∗ memcopy (void ∗ *dst,* const void ∗ *src,* uint32_t *count*)

copies bytes from a memory address to another memory address

Parameters

void∗	the destination pointer.
const	void∗ the source pointer.
uint32_t	the number of bytes to copy.

Returns

the destination pointer.

Definition at line 22 of file string.c.

4.7.2.16 memset (void ∗ *ptr,* int *c,* uint32_t *count*)

sets a number of bytes starting from a given pointer to a given character.

Parameters

void∗	the pointer to use.
int	character to set.
uint32_t	the number of bytes to set.

Returns

the pointer after being set.

Definition at line 34 of file string.c.

4.7.2.17 void play (void)

this function holds initializations and setting up facilities

Definition at line 175 of file play.c.

4.7.2.18 int printk (const char * *format*, ...)

An ideal printing function.

Parameters

const	char* format string, an ideal format string.
stdarg	argument list, variable referenced.

Returns

int number of chars written.

A printk function is simply a wrapper to the vprintk function. It reads the paramter list and pass it to vprintk.

Definition at line 202 of file printf.c.

4.7.2.19 static void putch (int *c*, int * *count*) [static]

prints a character at a time.

Parameters

int	character to write.
*int**	reference of count variable to increment per print.

Definition at line 23 of file printf.c.

4.7.2.20 void putchr (int *c*)

writes a character to screen.

Parameters

int	char to be written to display buffer.

Definition at line 61 of file cli.c.

4.7.2.21 char ∗ readline (const char ∗ *towrite*)

reads a string from keyboard into screen.

Parameters

const	char ∗ character to write

Returns

the string read.

It uses the put and get char functions from the cli code. then keep on receiving characters until [ENTER] is pressed.

Definition at line 22 of file readline.c.

4.7.2.22 int strcmp (const char ∗ *s1*, const char ∗ *s2*)

Definition at line 43 of file string.c.

4.7.2.23 uint32_t strnlen (const char ∗ *str*, uint32_t *count*)

the function loops over the string character values till '\0' is reached or reached max value.

Parameters

const	char∗ string.
uint32_t	the maximum size desired.

Returns

the number of chars in a string.

Definition at line 14 of file string.c.

4.7.2.24 void time_print (void)

a completely useless function that prints current time.

Definition at line 63 of file play.c.

4.7.2.25 int vprintk (const char ∗ *format*, va_list *ap*)

passes arugments to kvprintk after tokenizing them

Parameters

const	char* format string.
va_list	paramter list.

Returns

int number of chars written.

Definition at line 195 of file printf.c.

4.7.2.26 void work_it_out (uint32_t *mboot_ptr*)

Definition at line 35 of file work_it_out.c.

4.7.3 Variable Documentation

4.7.3.1 char buf[BUFLEN] `[static]`

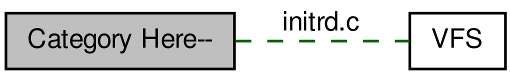
Definition at line 19 of file readline.c.

4.7.3.2 const char* error_panic = NULL `[static]`

Definition at line 40 of file play.c.

4.8 Category Here--

Collaboration diagram for Category Here--:



Files

- file initrd.c

  Initrd.

Initrd

- void ∗ initrd_mem ()

4.8.1 Function Documentation

4.8.1.1 void∗ initrd_mem ( )

Definition at line 15 of file initrd_mem.c.

4.9 Debug

Files

- file kconsole.h

    Kernel console for debugging ease.

KConsole

- kcommand_t ∗ kconsole_commands
- void kconsole_init (void)
- void kconsole (void)
- void kcommand_register (kcommand_t ∗cmd)
- kcommand_t ∗ kcommand_match (const char ∗buf)
- void kscheduler_info (void)
- void kcpu_info (void)
- void kconsole_help (void)

4.9.1 Function Documentation

4.9.1.1 kcommand_t∗ kcommand_match (const char ∗ *buf*)

Definition at line 74 of file kconsole.c.

4.9.1.2 void kcommand_register (kcommand_t ∗ *cmd*)

Definition at line 56 of file kconsole.c.

4.9.1.3 void kconsole (void)

Definition at line 36 of file kconsole.c.

4.9.1.4 void kconsole_help (void)

Definition at line 101 of file kconsole.c.

4.9.1.5 void kconsole_init (void)

Definition at line 16 of file kconsole.c.

4.9.1.6 void kcpu_info (void)

Definition at line 96 of file kconsole.c.

4.9.1.7 void kscheduler_info (void)

KCOMMANDS

Definition at line 91 of file kconsole.c.

4.9.2 Variable Documentation

4.9.2.1 kcommand_t* kconsole_commands

Definition at line 13 of file kconsole.c.

4.10 Management

Files

- file mm.c

Memory allocation/deallocation

Memory Management

Memory allocation is done by iniating an early hash table in which every page is stored as a bucket, within every bucket the size of memory allocated on that page sequentially. e.g. [Bucket:1]->[heap1:4]->[heap2:32]->[heap3:1024]->... in the first page 3 heaps are allocated 1st with size 4 bytes 2nd with size 32 byte and third with size 1Kb.

- htable_t ∗ mm_table
- uint32_t mm_va_base
- uint32_t cur_bucket
- void kmalloc_test (void)
- uint32_t mm_hash (uint32_t key)

    hash function for allocated memory entries in heaps hashtable
- uint32_t mm_destroy (uint32_t destroy)

    destroy function to free unwanted entries
- void mm_init (void)

    memory heap manager intialization function
- static uint32_t mm_insert (uint32_t va, uint32_t size)
- void ∗ kmalloc (uint32_t size)
- void kfree (void ∗va)

4.10.1 Function Documentation

4.10.1.1 void kfree (void ∗ *va*)

Parameters

void	∗va removes entry correspending to heap.

Returns

nothing.

Definition at line 153 of file mm.c.

4.10.1.2 void∗ kmalloc (uint32_t *size*)

Parameters

uint32_t	size, the size of desired heap.

Returns

fails Null, success returns heap base address.

Loops per each page and checks if it is already full. If full use next page for next bucket. else return base address.

Definition at line 132 of file mm.c.

4.10.1.3 void kmalloc_test (void)

Parameters

nothing	

Returns

nothing

tests abilities to remove and allocate heaps in healthy manner.

Definition at line 166 of file mm.c.

4.10.1.4 uint32_t mm_destroy (uint32_t *destroy*)

destroy function to free unwanted entries

Parameters

uint32_t	destroy is the entry or key to be destroyed

Returns

uint32_t returns 0 (indicating given entry is free)

a free entry is identifed by the value 0, to indicate that an entry is free value is simply left however a key is freed

Definition at line 59 of file mm.c.

4.10.1.5 uint32_t mm_hash (uint32_t *key*)

hash function for allocated memory entries in heaps hashtable

Parameters

uint32_t	key is the virtual address of table entry

Returns

uint32_t the returned hashed index of given key

since heaps are ordered and traced by a hashtable a hashing function to determine index of a given address into table is mandatory. such a function returns the index of page into page tables structures since in a chained hashtable heaps that belong to same page exist in the same linkedlist

Definition at line 46 of file mm.c.

4.10.1.6 void mm_init (void)

memory heap manager intialization function

Parameters

nothing	

Returns

nothing

For a heap manager to be initialized a boot time heap is allocated to hold heaps entries. Early allocated heap is mapped to kernel address space and virtual address base of heaps is set. Tests are exectued eventually.

Definition at line 76 of file mm.c.

4.10.1.7 static uint32_t mm_insert (uint32_t *va*, uint32_t *size*) [static]

Parameters

uint32_t	va, the desired heap virtual base address
uint32_t	size, the heap size to be allocated

Returns

fail -1, else return inserted base virtual address

on first use of a bucket a page is allocated then iteration is done until free address is reached then insertion to hash table is commited. On success it returns base address.

Definition at line 98 of file mm.c.

4.10.2 Variable Documentation

4.10.2.1 uint32_t cur_bucket

Definition at line 28 of file mm.c.

4.10.2.2 htable_t* mm_table

Here's the hash entry values 1- key = page number 2- val = size

Definition at line 26 of file mm.c.

4.10.2.3 uint32_t mm_va_base

Definition at line 27 of file mm.c.

4.11 Optimization

Files

- file mergesort.c

  Merge Sort implementation.
- file mergesort.c

  Merge Sort implementation.

Sorting(Merge)

- void mergesort (uint32_t array[ ], uint32_t temp[ ], uint32_t left, uint32_t right)
- void merge (uint32_t array[ ], uint32_t temp[ ], uint32_t left, uint32_t center, uint32-_t right)

Sorting(Merge)

- static void swap (int array[ ], int i, int j)
- static int partition (int array[ ], int low, int high)
- static void quick_sort (int array[ ], int low, int high)
- void quicksort (int array[ ], int size)
- int main (void)

4.11.1 Function Documentation

4.11.1.1 int main (void)

Definition at line 62 of file quicksort.c.

4.11.1.2 void merge (uint32_t *array[ ],* uint32_t *temp[ ],* uint32_t *left,* uint32_t *center,* uint32_t *right*)

Definition at line 23 of file mergesort.c.

4.11.1.3 void mergesort (uint32_t *array[ ],* uint32_t *temp[ ],* uint32_t *left,* uint32_t *right*)

Definition at line 12 of file mergesort.c.

4.11.1.4 static int partition (int *array[ ],* int *low,* int *high*) [static]

Definition at line 23 of file quicksort.c.

4.11.1.5 static void quick_sort (int *array[ ]*, int *low*, int *high*) [static]

Definition at line 47 of file quicksort.c.

4.11.1.6 void quicksort (int *array[ ]*, int *size*)

Definition at line 55 of file quicksort.c.

4.11.1.7 static void swap (int *array[ ]*, int *i*, int *j*) [static]

Definition at line 15 of file quicksort.c.

4.12 Process-Management

Files

- file load_elf.c

  Elf binary loader.
- file proc.c

  Process Manager.

Binary Loader.

- uint32_t elf_load_to_proc (proc_t *proc, uint32_t offset)

  loads a binary into a proc from an offset into image.

Process Manager.

- pde_t * global_pgdir
- struct Proc_List empty_procs
- proc_t * proc_table
- struct Proc_Lifo running_procs
- FIFO_HEAD (ready_procs, proc, 256)
- void proc_printinfo (void)

  Initialize the proc array.
- void init_proc_table (void)
- uint32_t create_proc (proc_t **proc_s)

  Initialize and create a proc information block.
- void proc_ready (proc_t *proc)
- uint32_t proc_alloc_mem (proc_t *proc, void *va, uint32_t len)

  allocates memory thunk for a proc.
- uint32_t init_proc0 ()

  loads and initiated the proc0(first proc) from image.
- void init_proc (void)

  initiates the process table and tests queues and initates proc0
- void test_lifo (void)

  Tests the LIFO queues data structure functionality.
- void test_fifo (void)

  Tests the FIFO queue data structure.
- void switch_address_space (proc_t *proc_to_run)

  switches between the kernel and a given proc
- void sched_init (void)
- void schedule (void)

  main scheduling function

4.12.1 Detailed Description

odefine FIFO_EXTERN(nme, member)\ extern struct fifo name;

4.12.2 Function Documentation

4.12.2.1 uint32_t create_proc (proc_t ** *proc_s*)

Initialize and create a proc information block.

Parameters

proc_t	proc pointer reference to store created proc in.

Returns

0 if success

a proc address space is initialized and segment desciptors are set, and the Stack is also set.

Definition at line 103 of file proc.c.

4.12.2.2 uint32_t elf_load_to_proc (proc_t * *proc,* uint32_t *offset*)

loads a binary into a proc from an offset into image.

Parameters

*proc_t**	proc to load binary into
uint32_t	offset of the binary into image

Returns

0 if success

The binary loader loads a binary image into the 0xA000000 VA where all procs elf are loaded.

Program headers are loaded into proc page directory and eip is set.

Definition at line 28 of file load_elf.c.

4.12.2.3 FIFO_HEAD (ready_procs , proc , 256)

4.12.2.4 void init_proc (void)

initiates the process table and tests queues and initates proc0

Definition at line 206 of file proc.c.

4.12.2.5 uint32_t init_proc0 ( )

loads and initiated the proc0(first proc) from image.

Definition at line 176 of file proc.c.

4.12.2.6 void init_proc_table (void)

Parameters

nothing	

Returns

nothing

Commits process holding structures (LIST/LIFO/FIFO) initalizes PEBs and address spaces.

Definition at line 57 of file proc.c.

4.12.2.7 uint32_t proc_alloc_mem (proc_t * *proc*, void * *va*, uint32_t *len*)

allocates memory thunk for a proc.

Parameters

proc_t	proc reference to allocate memory for
*void**	the base address of memory needed to be allocated.
uint32_t	the size of segment.

Returns

0 if success

Definition at line 154 of file proc.c.

4.12.2.8 void proc_printinfo (void)

Initialize the proc array.

Initializes all the procs data structures, Proc table, empty proc list and the scheduling process queues. All proc entries are set to 0's. and empty proc list is fill. and proc binary loader is mapped.

Definition at line 39 of file proc.c.

4.12.2.9 void proc_ready (proc_t * *proc*)

Parameters

proc_t∗	proc, process to use

Returns

nothing

this small process is used by scheduler to set a process as ready and push it into ready procs to be scheduled into ready_procs FIFO

Definition at line 140 of file proc.c.

4.12.2.10 void sched_init (void)

Definition at line 317 of file proc.c.

4.12.2.11 void schedule (void)

main scheduling function

Parameters

nothing	

Returns

nothing

a very basic RR scheduling function which detach running procs and push them into ready procs then fetch them. UNREACHABLE

Definition at line 333 of file proc.c.

4.12.2.12 void switch_address_space (proc_t ∗ *proc_to_run*)

switches between the kernel and a given proc

switching to a proc is made from a calling proc which is the kernel to a given proc address space. switching is done by Fooling the x86 into beleiving that it is coming back from an interrupt by setting the hardware interrupt stack frame and issuing an iret switching can also be made by SYSENTER/SYSEXIT.

Definition at line 269 of file proc.c.

4.12.2.13 void test_fifo (void)

Tests the FIFO queue data structure.

Definition at line 241 of file proc.c.

4.12.2.14 void test_lifo (void)

Tests the LIFO queues data structure functionality.

Definition at line 218 of file proc.c.

4.12.3 Variable Documentation

4.12.3.1 struct Proc_List empty_procs

Definition at line 25 of file proc.c.

4.12.3.2 pde_t* global_pgdir

Definition at line 25 of file page.c.

4.12.3.3 proc_t* proc_table

Definition at line 27 of file proc.c.

4.12.3.4 struct Proc_Lifo running_procs

Definition at line 28 of file proc.c.

4.13 Data-Structures

Files

- file htable.c

  Chained hash table.

Functions

- uint32_t early_htable_init (htable_t ∗table, uint32_t buckets_count, uint32_t bucket_length, uint32_t(∗hash)(uint32_t), uint32_t(∗destroy)(uint32_t))
- void htable_init (htable_t ∗table, uint32_t buckets_count, uint32_t(∗hash)(uint32_t), uint32_t(∗destroy)(uint32_t))

  Initalizes new hash table.
- void htable_destroy (htable_t ∗table)

  deletes hash table
- uint32_t htable_insert (htable_t ∗table, uint32_t key, uint32_t value)
- uint32_t htable_remove (htable_t ∗table, uint32_t key)

  Inserts value into the hash table.
- uint32_t htable_get (htable_t ∗table, uint32_t key)

  hash table look up function with O(n)

4.13.1 Function Documentation

4.13.1.1 uint32_t early_htable_init (htable_t ∗ *table*, uint32_t *buckets_count*, uint32_t *bucket_length*, uint32_t(∗)(uint32_t) *hash*, uint32_t(∗)(uint32_t) *destroy*)

Definition at line 23 of file htable.c.

4.13.1.2 htable_destroy (htable_t ∗ *table*)

deletes hash table

Parameters

htable_t∗	the reference to hash table

destroys hash tables buckets using the internal destroy function if not it just frees the memory the HTable is using Note: Early hash table SHOULD NOT be destroyed.

Definition at line 94 of file htable.c.

4.13.1.3 htable_get (htable_t ∗ *table*, uint32_t *key*)

hash table look up function with O(n)

Parameters

htable_t∗	reference to Hash table
uint32_t	key to return its value or -1 if not found

Definition at line 169 of file htable.c.

4.13.1.4 void htable_init (htable_t ∗ *table,* uint32_t *buckets_count,* uint32_t(∗)(uint32_t) *hash,* uint32_t(∗)(uint32_t) *destroy*)

Initalizes new hash table.

Parameters

htable_t∗	a non NULL hash-table pointer
uint32_t	the number of buckets to be initalized
uint32_-t(∗)(uint32_t)	hashing function
uint32_-t(∗)(uint32_t)	table destroy function

FIXME

allocate space for table inside the htable once kmalloc is made.

Definition at line 64 of file htable.c.

4.13.1.5 uint32_t htable_insert (htable_t ∗ *table,* uint32_t *key,* uint32_t *value*)

Definition at line 120 of file htable.c.

4.13.1.6 htable_remove (htable_t ∗ *table,* uint32_t *key*)

Inserts value into the hash table.

Parameters

htable_t∗	reference to hash table
uint32_t	the value to be stored.

Definition at line 146 of file htable.c.

4.14 Synchronization

Data Structures

- struct semaphore_t
- struct waiting_proc

Files

- file mutex.c
- file semaphore.c
- file wait_queue.c
- file semaphore.h

  Semaphore structures and function headers.

Typedefs

- typedef struct waiting_proc waiting_t

Functions

- LIST_HEAD (Wait_list, waiting_proc)

Mutex

Mutex synchronization primitives

- void mutex_init (semaphore_t ∗s)

  initialization to mutex structs and values.
- void mutex_lock (semaphore_t ∗s, proc_t ∗proc)

  blocks a process over a mutex
- void mutex_unlock (semaphore_t ∗s)

  sets binary resource to free

Semaphores

Semaphore synchronization primitives

- void semaphore_init (semaphore_t ∗s, uint32_t num)

  Initialization of a semaphore primitive function.
- void semaphore_down (semaphore_t ∗s, proc_t ∗proc)

  does down operation over semaphore primitive
- void semaphore_up (semaphore_t ∗s)

  Up function for a semaphore.

Wait queues

Wait queues

- void wait_init (struct Proc_List ∗waiting_procs)

  Waiting procs linked list initialization function.
- void wait_sleep (struct Proc_List ∗list, proc_t ∗proc, uint32_t ticks)

  function that blocks a process
- void wait_update (void)

  blocked process updating function
- void wait_wakeup (struct Proc_List ∗list)

  wakes up a proccess once timeout or resource free

4.14.1 Typedef Documentation

4.14.1.1 typedef struct waiting_proc waiting_t

4.14.2 Function Documentation

4.14.2.1 LIST_HEAD (Wait_list , waiting_proc)

4.14.2.2 void mutex_init (semaphore_t ∗ *s*)

initialization to mutex structs and values.

Parameters

s	the base semahpore of mutex to initalize.

Returns

nothing

A mutex is a binary semaphore thus, counting variable is set to one and a list of waiting processes is initialized.

Definition at line 20 of file mutex.c.

4.14.2.3 void mutex_lock (semaphore_t ∗ *s*, proc_t ∗ *proc*)

blocks a process over a mutex

Parameters

s	the semaphore that belongs to mutex
proc	the process to block

Returns

nothing

a binary clone to the semaphore down function.

Definition at line 34 of file mutex.c.

4.14.2.4 void mutex_unlock (semaphore_t * s)

sets binary resource to free

Parameters

s	mutex base semaphore

Returns

nothing

a clone of semaphore up function

Definition at line 49 of file mutex.c.

4.14.2.5 void semaphore_down (semaphore_t * s, proc_t * proc)

does down operation over semaphore primitive

Parameters

s	the semaphore to do down for.
proc	the process to block

aquire semaphore If a resource is available aquire it else wait with max time till waken up on resource release

Definition at line 35 of file semaphore.c.

4.14.2.6 void semaphore_init (semaphore_t * s, uint32_t num)

Initialization of a semaphore primitive function.

Parameters

s	the semaphore to initialize
num	the number of resources (counting/binary)

Returns

nothing

Initialization is simply done by setting semaphore count to given number and initialize the list of waiting procs

Definition at line 22 of file semaphore.c.

4.14.2.7 void semaphore_up (semaphore_t * *s*)

Up function for a semaphore.

Parameters

s	the semaphore to free resource for

Returns

nothing

Once a resource is free the semaphore counting variable is incrememnted if there's no waiting processes.. else a waiting process is brought out to take its place. release semaphore

Definition at line 58 of file semaphore.c.

4.14.2.8 void wait_init (struct Proc_List * *waiting_procs*)

Waiting procs linked list initialization function.

Parameters

waiting_-procs,list	to hold waiting procs pointer

Returns

nothing

Definition at line 20 of file wait_queue.c.

4.14.2.9 void wait_sleep (struct Proc_List * *list,* proc_t * *proc,* uint32_t *ticks*)

function that blocks a process

Parameters

list,the	sleeping processes list
proc,the	process to put to sleep
ticks,number	of ticks before wake-up

Returns

void

Generally, a sleep is done on an I/O or race conditions Here, a sleep is done either timed out or forever..relatively. and proc status is changed into blocked, inserted into waiting list and scheduler is set to reschedule procs.

Definition at line 38 of file wait_queue.c.

4.14.2.10 void wait_update (void)

blocked process updating function

Parameters

nothing	

Returns

nothing

For each timer tick/interrupt a wait_update is made to wake up timed out procs and update other waiting procs timers. __NOT_USED__

Definition at line 66 of file wait_queue.c.

4.14.2.11 void wait_wakeup (struct Proc_List * *list*)

wakes up a proccess once timeout or resource free

Parameters

list,the	blocked processes list

Returns

nothing

Basically, This function serves procs blocked due to a lock or semaphore..etc the first proc is marked as ready and pushed into ready procs

Definition at line 93 of file wait_queue.c.

4.15 Time

Files

- file periodic.c

  Periodic interrupt handler.

Periodic_Timer

- void time_handler (void)

4.15.1 Function Documentation

4.15.1.1 void time_handler (void)

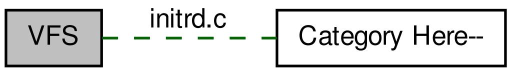
TODO

- Wake up sleeping procs if timed out
- update timers
- Execute scheduler

Definition at line 13 of file periodic.c.

4.16 VFS

Collaboration diagram for VFS:



Data Structures

- struct initrd_header_t
- struct initrd_file_header_t
- struct direntry_t
- struct vfs_node

Files

- file initrd.c

  Initrd.
- file initrd.h

  Initrd file system headers.
- file vfs.c

  Virtual file system standards , In order to abstract different file system calls.
- file vfs.h

  Standardized file system structures and constants.

Initrd

- static initrd_header_t ∗ initrd_header = NULL
- static initrd_file_header_t ∗ file_headers = NULL
- static vfs_node_t ∗ initrd_root = NULL
- static vfs_node_t ∗ initrd_dev = NULL
- static vfs_node_t ∗ root_nodes = NULL
- int nroot_nodes
- direntry_t dirent
- uint32_t initrd_read (vfs_node_t ∗node, uint32_t offset, uint32_t size, uint8_-t ∗buffer)

Read operation implementation , reads the offset from the node structure , Reads data of size specified by the call ; after checking that the file is legal.

- static direntry_t * initrd_readdir (vfs_node_t *node, uint32_t index)

  Directory traversal , reads next directory ; still immature.

- static vfs_node_t * initrd_finddir (vfs_node_t *node, char *name)

  Directory traversal , searches for a direcotry , very Immature.

- vfs_node_t * initialise_initrd (uint32_t location)

  Initializes file systems structures (root directory , /dev direcotry and file nodes) to be able to preform files system operations.

Initrd

- vfs_node_t * fs_root = 0
- uint32_t read_fs (vfs_node_t *node, uint32_t offset, uint32_t size, uint8_t *buffer)

  Calls the node's read operation , saved as read in the node structure.

- uint32_t write_fs (vfs_node_t *node, uint32_t offset, uint32_t size, uint8_t *buffer)

  Calls the node's write operation , saved as write in the node structure.

- uint8_t open_fs (vfs_node_t *node, uint8_t read, uint8_t write)

  Calls the node's open operation , saved as write in the node structure.

- uint8_t close_fs (vfs_node_t *node)

  Calls the node's open operation , saved as write in the node structure.

- direntry_t * readdir_fs (vfs_node_t *node, uint32_t index)

  Calls the node's read directory operation , saved as readdir in the node structure.

- vfs_node_t * finddir_fs (vfs_node_t *node, char *name)

  Calls the node's find directory operation , save as finddir in the node strucutre.

Initrd

- typedef uint32_t(* read_type_t)(struct vfs_node *, uint32_t, uint32_t, uint8_t *)
- typedef uint32_t(* write_type_t)(struct vfs_node *, uint32_t, uint32_t, uint8_t *)
- typedef uint8_t(* open_type_t)(struct vfs_node *, uint32_t, uint32_t)
- typedef uint8_t(* close_type_t)(struct vfs_node *)
- typedef direntry_t *(* readdir_type_t)(struct vfs_node *, uint32_t)
- typedef struct vfs_node *(* finddir_type_t)(struct vfs_node *, char *name)
- typedef struct vfs_node vfs_node_t
- #define FS_FILE 0x01
- #define FS_DIRECTORY 0x02
- #define FS_CHARDEVICE 0x03
- #define FS_BLOCKDEVICE 0x04
- #define FS_PIPE 0x05
- #define FS_SYMLINK 0x06
- #define FS_MOUNTPOINT 0x08

4.16.1 Define Documentation

4.16.1.1 #define FS_BLOCKDEVICE 0x04

Definition at line 18 of file vfs.h.

4.16.1.2 #define FS_CHARDEVICE 0x03

Definition at line 17 of file vfs.h.

4.16.1.3 #define FS_DIRECTORY 0x02

Definition at line 16 of file vfs.h.

4.16.1.4 #define FS_FILE 0x01

Definition at line 15 of file vfs.h.

4.16.1.5 #define FS_MOUNTPOINT 0x08

Definition at line 21 of file vfs.h.

4.16.1.6 #define FS_PIPE 0x05

Definition at line 19 of file vfs.h.

4.16.1.7 #define FS_SYMLINK 0x06

Definition at line 20 of file vfs.h.

4.16.2 Typedef Documentation

4.16.2.1 typedef uint8_t(* close_type_t)(struct vfs_node *)

Definition at line 28 of file vfs.h.

4.16.2.2 typedef struct vfs_node*(* finddir_type_t)(struct vfs_node *, char *name)

Definition at line 35 of file vfs.h.

4.16.2.3 typedef uint8_t(* open_type_t)(struct vfs_node *, uint32_t, uint32_t)

Definition at line 27 of file vfs.h.

4.16.2.4 typedef uint32_t(∗ read_type_t)(struct vfs_node ∗, uint32_t, uint32_t, uint8_t ∗)

Definition at line 25 of file vfs.h.

4.16.2.5 typedef direntry_t∗(∗ readdir_type_t)(struct vfs_node ∗, uint32_t)

Definition at line 34 of file vfs.h.

4.16.2.6 typedef struct vfs_node vfs_node_t

4.16.2.7 typedef uint32_t(∗ write_type_t)(struct vfs_node ∗, uint32_t, uint32_t, uint8_t ∗)

Definition at line 26 of file vfs.h.

4.16.3 Function Documentation

4.16.3.1 uint8_t close_fs (vfs_node_t ∗ *node*)

Calls the node's open operation , saved as write in the node structure.

Parameters

vfs_node_t	∗node the file's node to open.

Returns

uint32_t : 0 if the node does not support open operation else The result of the close operation.

Definition at line 74 of file vfs.c.

4.16.3.2 vfs_node_t∗ finddir_fs (vfs_node_t ∗ *node,* char ∗ *name*)

Calls the node's find directory operation , save as finddir in the node strucutre.

Parameters

vfs_node_t	∗node head to start from.
uint32_t	char ∗ name name of the directory to search for.

Returns

uint32_t : pointer to the node if succeeded , 0 if failed.

Definition at line 106 of file vfs.c.

4.16.3.3 vfs_node_t* initialise_initrd (uint32_t *location*)

Initializes file systems structures (root directory , /dev direcotry and file nodes) to be able to preform files system operations.

Parameters

uint32_t	location : pointer to the start of the initrd image.

Returns

vfs_node_t* initrd filesystem root node.

Definition at line 94 of file initrd.c.

4.16.3.4 static vfs_node_t* initrd_finddir (vfs_node_t * *node,* char * *name*) [static]

Directory traversal , searches for a direcotry , very Immature.

Parameters

vfs_node_t	*node head to start from.
uint32_t	char * name name of the directory to search for.

Returns

uint32_t : pointer to directory entry if succeeded , 0 if failed.

Definition at line 78 of file initrd.c.

4.16.3.5 uint32_t initrd_read (vfs_node_t * *node,* uint32_t *offset,* uint32_t *size,* uint8_t * *buffer*)

Read operation implementation , reads the offset from the node structure , Reads data of size specified by the call ; after checking that the file is legal.

Parameters

vfs_node_t	*node the file's node to be read.
uint32_t	offset where to start reading (0 for beggining of the file).
uint32_t	size size of data to read.
uint8_t	*buffer buffer to be read into.

Returns

uint32_t : 0 is failed , size of data read if succeeded .

Definition at line 32 of file initrd.c.

4.16.3.6 static direntry_t∗ initrd_readdir (vfs_node_t ∗ *node,* uint32_t *index*) [static]

Directory traversal , reads next directory ; still immature.

Parameters

vfs_node_t	∗node the directory node to be read.
uint32_t	directory index.

Returns

uint32_t : pointer to directory entry if succeeded , 0 if failed.

Definition at line 54 of file initrd.c.

4.16.3.7 uint8_t open_fs (vfs_node_t ∗ *node,* uint8_t *read,* uint8_t *write*)

Calls the node's open operation , saved as write in the node structure.

Parameters

vfs_node_t	∗node the file's node to open.
uint8_t	read read flag.
uint8_t	write write flag

Returns

uint32_t : 0 if the node does not support open operation else The result of the open operation.

Definition at line 60 of file vfs.c.

4.16.3.8 uint32_t read_fs (vfs_node_t ∗ *node,* uint32_t *offset,* uint32_t *size,* uint8_t ∗ *buffer*)

Calls the node's read operation , saved as read in the node structure.

Parameters

vfs_node_t	∗node the file's node to be read.
uint32_t	offset where to start reading (0 for beggining of the file).
uint32_t	size size of data to read.
uint8_t	∗buffer buffer to be read into.

Returns

uint32_t : 0 if the node does not support read operation else The result of the read operation.

Definition at line 27 of file vfs.c.

4.16.3.9 direntry_t* readdir_fs (vfs_node_t * *node*, uint32_t *index*)

Calls the node's read directory operation , saved as readdir in the node structure.

Parameters

vfs_node_t	*node the directory node to be read.
uint32_t	directory index.

Returns

direntry_t * : 0 if the node does not support open operation else The directory entry pointer returned by the readdir operation

Definition at line 89 of file vfs.c.

4.16.3.10 uint32_t write_fs (vfs_node_t * *node*, uint32_t *offset*, uint32_t *size*, uint8_t * *buffer*)

Calls the node's write operation , saved as write in the node structure.

Parameters

vfs_node_t	*node the file's node to be written into.
uint32_t	offset where to start write (0 for beggining of the file).
uint32_t	size size of data to write.
uint8_t	*buffer buffer containing data to write.

Returns

uint32_t : 0 if the node does not support open operation else The result of the write operation.

Definition at line 45 of file vfs.c.

4.16.4 Variable Documentation

4.16.4.1 direntry_t dirent

Definition at line 20 of file initrd.c.

4.16.4.2 initrd_file_header_t* file_headers = NULL [static]

Definition at line 15 of file initrd.c.

4.16.4.3 vfs_node_t* fs_root = 0

Definition at line 13 of file vfs.c.

4.16.4.4 vfs_node_t* initrd_dev = NULL [static]

Definition at line 17 of file initrd.c.

4.16.4.5 initrd_header_t* initrd_header = NULL [static]

Definition at line 14 of file initrd.c.

4.16.4.6 vfs_node_t* initrd_root = NULL [static]

Definition at line 16 of file initrd.c.

4.16.4.7 int nroot_nodes

Definition at line 19 of file initrd.c.

4.16.4.8 vfs_node_t* root_nodes = NULL [static]

Definition at line 18 of file initrd.c.

4.17 X86-boot

Data Structures

- struct elfhdr

  Elf binary header values.
- struct proghdr

  An ELF program header structure.
- struct sechdr

  An ELF section header structure.

Files

- file main.c

  Elf binary structures and constants, for parsing binaries.

Functions

- static __inline void outb (uint8_t data, int port)
- static __inline uint8_t inb (int port)
- static void outw (uint16_t data, int port)
- static uint16_t inw (int port)
- static __inline void outl (uint32_t data, int port)
- static uint32_t inl (int port)
- static void insb (void ∗addr, int cnt, int port)
- static void insl (void ∗addr, int cnt, int port)
- static void insw (void ∗addr, int cnt, int port)

Bootloader-KernelLoading

- void readsect (void ∗, uint32_t)

  reads disk at a given offset
- void readseg (uint32_t, uint32_t, uint32_t)

  loads data from disk into memory with data size to read.
- void cmain (void)

  reads and validates kernel image and loads it and fetch it into IP.
- void waitdisk (void)

  hangs till disk operation is done.
- #define SECTOR 512

  equals the sector size in bytes
- #define ELFHDR ((struct elfhdr ∗) 0x10000)

  The kernel physical address to load image into.

Elf-Member-Constants

- #define ELF_MAGIC 0x464C457F
- #define MAGIC_LEN 16
- #define M_CLASS_OFF 4
- #define M_CLASSNONE 0
- #define M_CLASS32 1
- #define M_CLASS64 2
- #define M_CLASSNUM 3
- #define M_DATA_OFF 5
- #define M_DATANONE 0
- #define M_DATA2LE 1
- #define M_DATA2BE 2
- #define M_DATANUM 3
- #define M_VERSION 6
- #define M_OSABI 7
- #define M_OSABI_SYSV 0
- #define M_OSABI_HPUX 1
- #define M_ABIVERSION 8
- #define M_ELF_PADDING 9
- #define T_TYPE_NONE 0
- #define T_TYPE_REL 1
- #define T_TYPE_EXEC 2
- #define T_TYPE_DYN 3
- #define T_TYPE_CORE 4
- #define T_TYPE_LOPROC 0xff00
- #define T_TYPE_HIPROC 0xffff
- #define M_MACHINE_I386 3
- #define V_VERSION_NONE 0
- #define V_VERSION_CURRENT 1
- #define V_VERSION_NUM 2
- #define P_PROGHDR_R 0x4
- #define P_PROGHDR_W 0x2
- #define P_PROGHDR_E 0x1

Elf-structures

- typedef struct elfhdr elfhdr
- typedef struct proghdr prohdr
- typedef struct sechdr secthdr

4.17.1 Define Documentation

4.17.1.1 #define ELF_MAGIC 0x464C457F

Definition at line 21 of file elf.h.

4.17.1.2 #define ELFHDR ((struct elfhdr *) 0x10000)

The kernel physical address to load image into.

Definition at line 37 of file main.c.

4.17.1.3 #define M_ABIVERSION 8

Definition at line 42 of file elf.h.

4.17.1.4 #define M_CLASS32 1

Definition at line 26 of file elf.h.

4.17.1.5 #define M_CLASS64 2

Definition at line 27 of file elf.h.

4.17.1.6 #define M_CLASS_OFF 4

Definition at line 24 of file elf.h.

4.17.1.7 #define M_CLASSNONE 0

Definition at line 25 of file elf.h.

4.17.1.8 #define M_CLASSNUM 3

Definition at line 28 of file elf.h.

4.17.1.9 #define M_DATA2BE 2

Definition at line 33 of file elf.h.

4.17.1.10 #define M_DATA2LE 1

Definition at line 32 of file elf.h.

4.17.1.11 #define M_DATA_OFF 5

Definition at line 30 of file elf.h.

4.17.1.12 #define M_DATANONE 0

Definition at line 31 of file elf.h.

4.17.1.13 #define M_DATANUM 3

Definition at line 34 of file elf.h.

4.17.1.14 #define M_ELF_PADDING 9

Definition at line 43 of file elf.h.

4.17.1.15 #define M_MACHINE_I386 3

Definition at line 59 of file elf.h.

4.17.1.16 #define M_OSABI 7

Definition at line 38 of file elf.h.

4.17.1.17 #define M_OSABI_HPUX 1

Definition at line 40 of file elf.h.

4.17.1.18 #define M_OSABI_SYSV 0

Definition at line 39 of file elf.h.

4.17.1.19 #define M_VERSION 6

Definition at line 36 of file elf.h.

4.17.1.20 #define MAGIC_LEN 16

Definition at line 22 of file elf.h.

4.17.1.21 #define P_PROGHDR_E 0x1

Definition at line 73 of file elf.h.

4.17.1.22 #define P_PROGHDR_R 0x4

Definition at line 71 of file elf.h.

4.17.1.23 #define P_PROGHDR_W 0x2

Definition at line 72 of file elf.h.

4.17.1.24 #define SECTOR 512

equals the sector size in bytes

Definition at line 32 of file main.c.

4.17.1.25 #define T_TYPE_CORE 4

Definition at line 52 of file elf.h.

4.17.1.26 #define T_TYPE_DYN 3

Definition at line 51 of file elf.h.

4.17.1.27 #define T_TYPE_EXEC 2

Definition at line 50 of file elf.h.

4.17.1.28 #define T_TYPE_HIPROC 0xffff

Definition at line 54 of file elf.h.

4.17.1.29 #define T_TYPE_LOPROC 0xff00

Definition at line 53 of file elf.h.

4.17.1.30 #define T_TYPE_NONE 0

Definition at line 48 of file elf.h.

4.17.1.31 #define T_TYPE_REL 1

Definition at line 49 of file elf.h.

4.17.1.32 #define V_VERSION_CURRENT 1

Definition at line 65 of file elf.h.

4.17.1.33 #define V_VERSION_NONE 0

Definition at line 64 of file elf.h.

4.17.1.34 #define V_VERSION_NUM 2

Definition at line 66 of file elf.h.

4.17.2 Typedef Documentation

4.17.2.1 typedef struct elfhdr elfhdr

4.17.2.2 typedef struct proghdr prohdr

4.17.2.3 typedef struct sechdr secthdr

4.17.3 Function Documentation

4.17.3.1 void cmain (void)

reads and validates kernel image and loads it and fetch it into IP.

Returns

nothing.

this function is called by the boot.S after setting up segmentation and operating in protected mode. it loads the kernel image and validate that it's an ELF image, then loads the program headers into memory. and jump to kernel image, if ever a kernel return it will spin or hang. and such thing is merely impossible. initialize magic to the first for characters of ELF MAGIC signature

Definition at line 74 of file main.c.

4.17.3.2 static __inline uint8_t inb (int *port*) `[static]`

Definition at line 21 of file x86.h.

4.17.3.3 static uint32_t inl (int *port*) `[inline, static]`

Definition at line 43 of file x86.h.

4.17.3.4 static void insb (void * *addr*, int *cnt*, int *port*) [inline, static]

Definition at line 49 of file x86.h.

4.17.3.5 static void insl (void * *addr*, int *cnt*, int *port*) [inline, static]

Definition at line 55 of file x86.h.

4.17.3.6 static void insw (void * *addr*, int *cnt*, int *port*) [inline, static]

Definition at line 61 of file x86.h.

4.17.3.7 static uint16_t inw (int *port*) [inline, static]

Definition at line 32 of file x86.h.

4.17.3.8 static __inline void outb (uint8_t *data*, int *port*) [static]

Definition at line 17 of file x86.h.

4.17.3.9 static __inline void outl (uint32_t *data*, int *port*) [static]

Definition at line 38 of file x86.h.

4.17.3.10 static void outw (uint16_t *data*, int *port*) [inline, static]

Definition at line 27 of file x86.h.

4.17.3.11 void readsect (void * *dst*, uint32_t *offset*)

reads disk at a given offset

Parameters

*void**	va, is the address to load read data into
uint32_t	offset, offset in image

Returns

nothing.

Definition at line 36 of file rdisk.c.

4.17.3.12 void readseg (uint32_t *va,* uint32_t *count,* uint32_t *offset*)

loads data from disk into memory with data size to read.

Parameters

uint32_t	va, virtual address to load data into
uint32_t	count, the number of sectors to load into memory.
uint32_t	offset, the offset of data into disk.

Returns

nothing.

It could be called as a wrapper function to readsect, since readseg uses readsect int its operations.

Definition at line 11 of file rdisk.c.

4.17.3.13 void waitdisk (void)

hangs till disk operation is done.

Returns

nothing.

this function just reads the disk port status register holds the value of 0x40 which means disk is not executing a command and ready.

Definition at line 110 of file main.c.

Chapter 5

Data Structure Documentation

5.1 __attribute Struct Reference

```
#include <multiboot.h>
```

Data Fields

- uint32_t flags
- uint32_t mem_lower
- uint32_t mem_upper
- uint32_t boot_device
- uint32_t cmdline
- uint32_t mods_count
- uint32_t mods_addr
- union {
    multiboot_aout_symbol_table_t aout_sym
    multiboot_elf_section_header_t elf_sec
  } u

- uint32_t mmap_length
- uint32_t mmap_addr
- uint32_t drives_length
- uint32_t drives_addr
- uint32_t config_table
- uint32_t boot_loader_name
- uint32_t apm_table
- uint32_t vbe_control_info
- uint32_t vbe_mode_info
- uint16_t vbe_mode
- uint16_t vbe_interface_seg
- uint16_t vbe_interface_off
- uint16_t vbe_interface_len

5.1.1 Detailed Description

Definition at line 82 of file multiboot.h.

5.1.2 Field Documentation

5.1.2.1 multiboot_aout_symbol_table_t __attribute::aout_sym

Definition at line 98 of file multiboot.h.

5.1.2.2 uint32_t __attribute::apm_table

Definition at line 112 of file multiboot.h.

5.1.2.3 uint32_t __attribute::boot_device

Definition at line 90 of file multiboot.h.

5.1.2.4 uint32_t __attribute::boot_loader_name

Definition at line 110 of file multiboot.h.

5.1.2.5 uint32_t __attribute::cmdline

Definition at line 92 of file multiboot.h.

5.1.2.6 uint32_t __attribute::config_table

Definition at line 108 of file multiboot.h.

5.1.2.7 uint32_t __attribute::drives_addr

Definition at line 106 of file multiboot.h.

5.1.2.8 uint32_t __attribute::drives_length

Definition at line 105 of file multiboot.h.

5.1.2.9 multiboot_elf_section_header_t __attribute::elf_sec

Definition at line 99 of file multiboot.h.

5.1.2.10 uint32_t __attribute::flags

Definition at line 85 of file multiboot.h.

5.1.2.11 uint32_t __attribute::mem_lower

Definition at line 87 of file multiboot.h.

5.1.2.12 uint32_t __attribute::mem_upper

Definition at line 88 of file multiboot.h.

5.1.2.13 uint32_t __attribute::mmap_addr

Definition at line 103 of file multiboot.h.

5.1.2.14 uint32_t __attribute::mmap_length

Definition at line 102 of file multiboot.h.

5.1.2.15 uint32_t __attribute::mods_addr

Definition at line 95 of file multiboot.h.

5.1.2.16 uint32_t __attribute::mods_count

Definition at line 94 of file multiboot.h.

5.1.2.17 union { ... } __attribute::u

5.1.2.18 uint32_t __attribute::vbe_control_info

Definition at line 114 of file multiboot.h.

5.1.2.19 uint16_t __attribute::vbe_interface_len

Definition at line 119 of file multiboot.h.

5.1.2.20 uint16_t __attribute::vbe_interface_off

Definition at line 118 of file multiboot.h.

5.1.2.21 uint16_t __attribute::vbe_interface_seg

Definition at line 117 of file multiboot.h.

5.1.2.22 uint16_t __attribute::vbe_mode

Definition at line 116 of file multiboot.h.

5.1.2.23 uint32_t __attribute::vbe_mode_info

Definition at line 115 of file multiboot.h.

The documentation for this struct was generated from the following file:

- include/multiboot.h

5.2 __attribute__ Struct Reference

Task State segment.

```
#include <cpu_state.h>
```

Data Fields

- uint16_t prelink
- uint16_t _rsrvd1
- reg_t esp0
- uint16_t ss0
- uint16_t _rsrvd2
- reg_t esp1
- uint16_t ss1
- uint16_t _rsrvd3
- reg_t esp2
- uint16_t ss2
- uint16_t _rsrvd4
- uint32_t cr3
- reg_t eip
- reg_t eflags
- reg_t eax
- reg_t ecx
- reg_t edx
- reg_t ebx
- reg_t esp
- reg_t ebp
- reg_t esi

- reg_t edi
- uint16_t es
- uint16_t _rsrvd5
- uint16_t cs
- uint16_t _rsrvd6
- uint16_t ss
- uint16_t _rsrvd7
- uint16_t ds
- uint16_t _rsrvd8
- uint16_t fs
- uint16_t _rsrvd9
- uint16_t gs
- uint16_t _rsrvda
- uint16_t ldt
- uint16_t _rsrvdb
- uint16_t trace
- uint16_t iomap_base
- uint16_t limit_0_15
- uint16_t base_0_15
- uint8_t base_16_23
- uint8_t access
- unsigned limit_16_19:4
- unsigned available:1
- unsigned unused:2
- unsigned granularity:1
- uint8_t base_24_31
- uint16_t offset_0_15
- uint16_t segment_s
- unsigned args:5
- unsigned reserved:3
- unsigned type:4
- unsigned s:1
- unsigned dpl:2
- unsigned p:1
- uint16_t offset_16_31
- unsigned present:1
- unsigned writable:1
- unsigned accessible:1
- unsigned write_through:1
- unsigned cache_disable:1
- unsigned accessed:1
- unsigned dirty:1
- unsigned pat:1
- unsigned global:1
- unsigned ignore_this:3
- unsigned address:20

- union {
  uint32_t lo
  };
- union {
  uint32_t hi
  };
- uint32_t Signature
- uintptr_t config_addr
- uint8_t len
- uint8_t version
- uint8_t checksum
- uint8_t feature1
- uint8_t feature2
- uint16_t base_table_len
- uint8_t spec_ver
- uint8_t oem_id [8]
- uint8_t product_id [12]
- uintptr_t oem_table_pointer
- uint16_t oem_table_size
- uint16_t entry_count
- uint32_t lapic_addr
- uint16_t extnd_table_len
- uint8_t extnd_table_checksum
- uint8_t reserved
- uint8_t base_table [0]
- uint8_t entry_type
- uint8_t lapic_id
- uint8_t lapic_version
- uint8_t cpu_flags
- uint8_t cpu_signature [4]
- uint32_t feature_flags
- uint8_t bus_id
- uint8_t bus_type [6]
- uint8_t io_apic_id
- uint8_t io_apic_ver
- uint8_t io_apic_flags
- uint8_t io_apic_addr
- uint8_t type
- uint8_t intrrupt_type
- uint8_t po_el
- uint8_t pad
- uint8_t src_bus_id
- uint8_t src_bus_irq
- uint8_t dst_io_apic_id
- uint8_t dst_io_apic_pin

- uint8_t dst_loc_apic_id
- uint8_t dst_loc_apic_pin
- uint32_t magic
- uint32_t flags
- uint32_t checksum
- uint32_t header_addr
- uint32_t load_addr
- uint32_t load_end_addr
- uint32_t bss_end_addr
- uint32_t entry_addr
- uint32_t mode_type
- uint32_t width
- uint32_t height
- uint32_t depth
- uint32_t num
- uint32_t size
- uint32_t addr
- uint32_t shndx
- uint32_t tabsize
- uint32_t strsize
- uint32_t reserved
- uint32_t moduleStart
- uint32_t moduleEnd
- char string [8]
- uint64_t addr
- uint64_t len
- uint32_t type
- uint32_t mod_start
- uint32_t mod_end
- uint32_t cmdline
- uint32_t pad
- uint8_t vector
- unsigned int delivery_mode: 3
- unsigned int dest_mode: 1
- unsigned int delivery_stat: 1
- unsigned int: 1
- unsigned int level: 1
- unsigned int trigger_mode: 1
- unsigned int shorthand: 2
- uint8_t dest

5.2.1 Detailed Description

Task State segment.

Task state segment descirptor.

The Task state segment entry fields, merely used into the kernel.

TSS is only used to support interrupts from user space. other than that soft task switching is used.

TSS desciptor is somehow a child of a gate desciptor and similar to segment selectors.

Gate descriptors as defined by Intel Manuals.

Definition at line 109 of file cpu_state.h.

5.2.2 Field Documentation

5.2.2.1 union { ... }

5.2.2.2 union { ... }

5.2.2.3 uint16_t __attribute__::_rsrvd1

Definition at line 110 of file cpu_state.h.

5.2.2.4 uint16_t __attribute__::_rsrvd2

Definition at line 112 of file cpu_state.h.

5.2.2.5 uint16_t __attribute__::_rsrvd3

Definition at line 114 of file cpu_state.h.

5.2.2.6 uint16_t __attribute__::_rsrvd4

Definition at line 116 of file cpu_state.h.

5.2.2.7 uint16_t __attribute__::_rsrvd5

Definition at line 128 of file cpu_state.h.

5.2.2.8 uint16_t __attribute__::_rsrvd6

Definition at line 129 of file cpu_state.h.

5.2.2.9 uint16_t __attribute__::_rsrvd7

Definition at line 130 of file cpu_state.h.

5.2.2.10 uint16_t __attribute__::_rsrvd8

Definition at line 131 of file cpu_state.h.

5.2.2.11 uint16_t __attribute__::_rsrvd9

Definition at line 132 of file cpu_state.h.

5.2.2.12 uint16_t __attribute__::_rsrvda

Definition at line 133 of file cpu_state.h.

5.2.2.13 uint16_t __attribute__::_rsrvdb

Definition at line 134 of file cpu_state.h.

5.2.2.14 uint8_t __attribute__::access

Definition at line 150 of file cpu_state.h.

5.2.2.15 unsigned __attribute__::accessed

Definition at line 42 of file page.h.

5.2.2.16 unsigned __attribute__::accessible

Definition at line 39 of file page.h.

5.2.2.17 uint32_t __attribute__::addr

Definition at line 62 of file multiboot.h.

5.2.2.18 uint64_t __attribute__::addr

Definition at line 125 of file multiboot.h.

5.2.2.19 unsigned __attribute__::address

Definition at line 47 of file page.h.

5.2.2.20 unsigned __attribute__::args

Definition at line 29 of file interrupt.h.

5.2.2.21 unsigned __attribute__::available

Definition at line 152 of file cpu_state.h.

5.2.2.22 uint16_t __attribute__::base_0_15

Definition at line 148 of file cpu_state.h.

5.2.2.23 uint8_t __attribute__::base_16_23

Definition at line 149 of file cpu_state.h.

5.2.2.24 uint8_t __attribute__::base_24_31

Definition at line 155 of file cpu_state.h.

5.2.2.25 uint8_t __attribute__::base_table[0]

Definition at line 65 of file smp.h.

5.2.2.26 uint16_t __attribute__::base_table_len

Definition at line 52 of file smp.h.

5.2.2.27 uint32_t __attribute__::bss_end_addr

Definition at line 47 of file multiboot.h.

5.2.2.28 uint8_t __attribute__::bus_id

Definition at line 83 of file smp.h.

5.2.2.29 uint8_t __attribute__::bus_type[6]

Definition at line 84 of file smp.h.

5.2.2.30 unsigned __attribute__::cache_disable

Definition at line 41 of file page.h.

5.2.2.31 uint32_t __attribute__::checksum

Definition at line 42 of file multiboot.h.

5.2.2.32 uint8_t __attribute__::checksum

Definition at line 43 of file smp.h.

5.2.2.33 uint32_t __attribute__::cmdline

Definition at line 138 of file multiboot.h.

5.2.2.34 uintptr_t __attribute__::config_addr

Definition at line 40 of file smp.h.

5.2.2.35 uint8_t __attribute__::cpu_flags

Definition at line 73 of file smp.h.

5.2.2.36 uint8_t __attribute__::cpu_signature[4]

Definition at line 77 of file smp.h.

5.2.2.37 uint32_t __attribute__::cr3

Definition at line 117 of file cpu_state.h.

5.2.2.38 uint16_t __attribute__::cs

Definition at line 129 of file cpu_state.h.

5.2.2.39 unsigned int __attribute__::delivery_mode

Definition at line 81 of file apic.h.

5.2.2.40 unsigned int __attribute__::delivery_stat

Definition at line 83 of file apic.h.

5.2.2.41 uint32_t __attribute__::depth

Definition at line 54 of file multiboot.h.

5.2.2.42 uint8_t __attribute__::dest

Definition at line 96 of file apic.h.

5.2.2.43 unsigned int __attribute__::dest_mode

Definition at line 82 of file apic.h.

5.2.2.44 unsigned __attribute__::dirty

Definition at line 43 of file page.h.

5.2.2.45 unsigned __attribute__::dpl

Definition at line 41 of file interrupt.h.

5.2.2.46 uint16_t __attribute__::ds

Definition at line 131 of file cpu_state.h.

5.2.2.47 uint8_t __attribute__::dst_io_apic_id

Definition at line 103 of file smp.h.

5.2.2.48 uint8_t __attribute__::dst_io_apic_pin

Definition at line 104 of file smp.h.

5.2.2.49 uint8_t __attribute__::dst_loc_apic_id

Definition at line 114 of file smp.h.

5.2.2.50 uint8_t __attribute__::dst_loc_apic_pin

Definition at line 115 of file smp.h.

5.2.2.51 reg_t __attribute__::eax

Definition at line 120 of file cpu_state.h.

5.2.2.52 reg_t __attribute__::ebp

Definition at line 125 of file cpu_state.h.

5.2.2.53 reg_t __attribute__::ebx

Definition at line 123 of file cpu_state.h.

5.2.2.54 reg_t __attribute__::ecx

Definition at line 121 of file cpu_state.h.

5.2.2.55 reg_t __attribute__::edi

Definition at line 127 of file cpu_state.h.

5.2.2.56 reg_t __attribute__::edx

Definition at line 122 of file cpu_state.h.

5.2.2.57 reg_t __attribute__::eflags

Definition at line 119 of file cpu_state.h.

5.2.2.58 reg_t __attribute__::eip

Definition at line 118 of file cpu_state.h.

5.2.2.59 uint32_t __attribute__::entry_addr

Definition at line 48 of file multiboot.h.

5.2.2.60 uint16_t __attribute__::entry_count

Definition at line 59 of file smp.h.

5.2.2.61 uint8_t __attribute__::entry_type

Definition at line 70 of file smp.h.

5.2.2.62 uint16_t __attribute__::es

Definition at line 128 of file cpu_state.h.

5.2.2.63 reg_t __attribute__::esi

Definition at line 126 of file cpu_state.h.

5.2.2.64 reg_t __attribute__::esp

Definition at line 124 of file cpu_state.h.

5.2.2.65 reg_t __attribute__::esp0

Definition at line 111 of file cpu_state.h.

5.2.2.66 reg_t __attribute__::esp1

Definition at line 113 of file cpu_state.h.

5.2.2.67 reg_t __attribute__::esp2

Definition at line 115 of file cpu_state.h.

5.2.2.68 uint8_t __attribute__::extnd_table_checksum

Definition at line 62 of file smp.h.

5.2.2.69 uint16_t __attribute__::extnd_table_len

Definition at line 61 of file smp.h.

5.2.2.70 uint8_t __attribute__::feature1

Definition at line 44 of file smp.h.

5.2.2.71 uint8_t __attribute__::feature2

Definition at line 45 of file smp.h.

5.2.2.72 uint32_t __attribute__::feature_flags

Definition at line 78 of file smp.h.

5.2.2.73 uint32_t __attribute__::flags

Definition at line 40 of file multiboot.h.

5.2.2.74 uint16_t __attribute__::fs

Definition at line 132 of file cpu_state.h.

5.2.2.75 unsigned __attribute__::global

Definition at line 45 of file page.h.

5.2.2.76 unsigned __attribute__::granularity

Definition at line 154 of file cpu_state.h.

5.2.2.77 uint16_t __attribute__::gs

Definition at line 133 of file cpu_state.h.

5.2.2.78 uint32_t __attribute__::header_addr

Definition at line 44 of file multiboot.h.

5.2.2.79 uint32_t __attribute__::height

Definition at line 53 of file multiboot.h.

5.2.2.80 uint32_t __attribute__::hi

Definition at line 93 of file apic.h.

5.2.2.81 unsigned __attribute__::ignore_this

Definition at line 46 of file page.h.

5.2.2.82 unsigned __attribute__::int

Definition at line 84 of file apic.h.

5.2.2.83 uint8_t __attribute__::intrrupt_type

Definition at line 98 of file smp.h.

5.2.2.84 uint8_t __attribute__::io_apic_addr

Definition at line 93 of file smp.h.

5.2.2.85 uint8_t __attribute__::io_apic_flags

Definition at line 92 of file smp.h.

5.2.2.86 uint8_t __attribute__::io_apic_id

Definition at line 90 of file smp.h.

5.2.2.87 uint8_t __attribute__::io_apic_ver

Definition at line 91 of file smp.h.

5.2.2.88 uint16_t __attribute__::iomap_base

Definition at line 136 of file cpu_state.h.

5.2.2.89 uint32_t __attribute__::lapic_addr

Definition at line 60 of file smp.h.

5.2.2.90 uint8_t __attribute__::lapic_id

Definition at line 71 of file smp.h.

5.2.2.91 uint8_t __attribute__::lapic_version

Definition at line 72 of file smp.h.

5.2.2.92 uint16_t __attribute__::ldt

Definition at line 134 of file cpu_state.h.

5.2.2.93 uint8_t __attribute__::len

Definition at line 41 of file smp.h.

5.2.2.94 uint64_t __attribute__::len

Definition at line 126 of file multiboot.h.

5.2.2.95 unsigned int __attribute__::level

Definition at line 85 of file apic.h.

5.2.2.96 uint16_t __attribute__::limit_0_15

Definition at line 147 of file cpu_state.h.

5.2.2.97 unsigned __attribute__::limit_16_19

Definition at line 151 of file cpu_state.h.

5.2.2.98 uint32_t __attribute__::lo

Definition at line 78 of file apic.h.

5.2.2.99 uint32_t __attribute__::load_addr

Definition at line 45 of file multiboot.h.

5.2.2.100 uint32_t __attribute__::load_end_addr

Definition at line 46 of file multiboot.h.

5.2.2.101 uint32_t __attribute__::magic

Definition at line 39 of file multiboot.h.

5.2.2.102 uint32_t __attribute__::mod_end

Definition at line 136 of file multiboot.h.

5.2.2.103 uint32_t __attribute__::mod_start

Definition at line 135 of file multiboot.h.

5.2.2.104 uint32_t __attribute__::mode_type

Definition at line 51 of file multiboot.h.

5.2.2.105 uint32_t __attribute__::moduleEnd

Definition at line 78 of file multiboot.h.

5.2.2.106 uint32_t __attribute__::moduleStart

Definition at line 77 of file multiboot.h.

5.2.2.107 uint32_t __attribute__::num

Definition at line 60 of file multiboot.h.

5.2.2.108 uint8_t __attribute__::oem_id[8]

Definition at line 55 of file smp.h.

5.2.2.109 uintptr_t __attribute__::oem_table_pointer

Definition at line 57 of file smp.h.

5.2.2.110 uint16_t __attribute__::oem_table_size

Definition at line 58 of file smp.h.

5.2.2.111 uint16_t __attribute__::offset_0_15

Definition at line 22 of file interrupt.h.

5.2.2.112 uint16_t __attribute__::offset_16_31

Definition at line 43 of file interrupt.h.

5.2.2.113 unsigned __attribute__::p

Definition at line 42 of file interrupt.h.

5.2.2.114 uint8_t __attribute__::pad

Definition at line 100 of file smp.h.

5.2.2.115 uint32_t __attribute__::pad

Definition at line 140 of file multiboot.h.

5.2.2.116 unsigned __attribute__::pat

Definition at line 44 of file page.h.

5.2.2.117 uint8_t __attribute__::po_el

Definition at line 99 of file smp.h.

5.2.2.118 uint16_t __attribute__::prelink

Definition at line 110 of file cpu_state.h.

5.2.2.119 unsigned __attribute__::present

Definition at line 37 of file page.h.

5.2.2.120 uint8_t __attribute__::product_id[12]

Definition at line 56 of file smp.h.

5.2.2.121 unsigned __attribute__::reserved

Definition at line 30 of file interrupt.h.

5.2.2.122 uint8_t __attribute__::reserved

Definition at line 64 of file smp.h.

5.2.2.123 uint32_t __attribute__::reserved

Definition at line 71 of file multiboot.h.

5.2.2.124 unsigned __attribute__::s

Definition at line 40 of file interrupt.h.

5.2.2.125 uint16_t __attribute__::segment_s

Definition at line 23 of file interrupt.h.

5.2.2.126 uint32_t __attribute__::shndx

Definition at line 63 of file multiboot.h.

5.2.2.127 unsigned int __attribute__::shorthand

Definition at line 88 of file apic.h.

5.2.2.128 uint32_t __attribute__::Signature

Definition at line 39 of file smp.h.

5.2.2.129 uint32_t __attribute__::size

Definition at line 61 of file multiboot.h.

5.2.2.130 uint8_t __attribute__::spec_ver

Definition at line 53 of file smp.h.

5.2.2.131 uint8_t __attribute__::src_bus_id

Definition at line 101 of file smp.h.

5.2.2.132 uint8_t __attribute__::src_bus_irq

Definition at line 102 of file smp.h.

5.2.2.133 uint16_t __attribute__::ss

Definition at line 130 of file cpu_state.h.

5.2.2.134 uint16_t __attribute__::ss0

Definition at line 112 of file cpu_state.h.

5.2.2.135 uint16_t __attribute__::ss1

Definition at line 114 of file cpu_state.h.

5.2.2.136 uint16_t __attribute__::ss2

Definition at line 116 of file cpu_state.h.

5.2.2.137 char __attribute__::string[8]

Definition at line 79 of file multiboot.h.

5.2.2.138 uint32_t __attribute__::strsize

Definition at line 69 of file multiboot.h.

5.2.2.139 uint32_t __attribute__::tabsize

Definition at line 68 of file multiboot.h.

5.2.2.140 uint16_t __attribute__::trace

Definition at line 135 of file cpu_state.h.

5.2.2.141 unsigned int __attribute__::trigger_mode

Definition at line 86 of file apic.h.

5.2.2.142 uint8_t __attribute__::type

Definition at line 39 of file interrupt.h.

5.2.2.143 uint8_t __attribute__::type

Definition at line 97 of file smp.h.

5.2.2.144 uint32_t __attribute__::type

Definition at line 129 of file multiboot.h.

5.2.2.145 unsigned __attribute__::unused

Definition at line 153 of file cpu_state.h.

5.2.2.146 uint8_t __attribute__::vector

Definition at line 80 of file apic.h.

5.2.2.147 uint8_t __attribute__::version

Definition at line 42 of file smp.h.

5.2.2.148 uint32_t __attribute__::width

Definition at line 52 of file multiboot.h.

5.2.2.149 unsigned __attribute__::writable

Definition at line 38 of file page.h.

5.2.2.150 unsigned __attribute__::write_through

Definition at line 40 of file page.h.

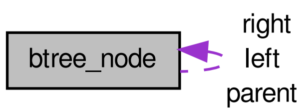
The documentation for this struct was generated from the following files:

- include/arch/x86/cpu_state.h
- include/arch/x86/interrupt.h
- include/arch/x86/mm/page.h
- include/arch/x86/mp/apic.h
- include/arch/x86/mp/smp.h
- include/multiboot.h

5.3 btree_node Struct Reference

`#include <btree.h>`

Collaboration diagram for btree_node:



Data Fields

- struct btree_node * parent
- struct btree_node * left
- struct btree_node * right
- uint32_t key
- uint32_t val

5.3.1 Detailed Description

Definition at line 9 of file btree.h.

5.3.2 Field Documentation

5.3.2.1 uint32_t btree_node::key

Definition at line 13 of file btree.h.

5.3.2.2 struct btree_node* btree_node::left

Definition at line 11 of file btree.h.

5.3.2.3 struct btree_node* btree_node::parent

Definition at line 10 of file btree.h.

5.3.2.4 struct btree_node* btree_node::right

Definition at line 12 of file btree.h.

5.3.2.5 uint32_t btree_node::val

Definition at line 14 of file btree.h.

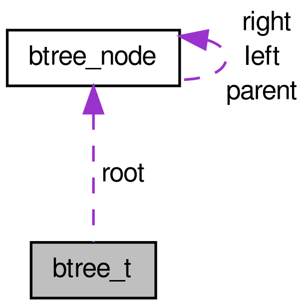
The documentation for this struct was generated from the following file:

- include/structs/btree.h

5.4 btree_t Struct Reference

```
#include <btree.h>
```

Collaboration diagram for btree_t:



Public Member Functions

- LIST_ENTRY (btree_t) link

Data Fields

- btree_node_t * root
- uint32_t size

5.4.1 Detailed Description

Definition at line 17 of file btree.h.

5.4.2 Member Function Documentation

5.4.2.1 btree_t::LIST_ENTRY (btree_t)

5.4.3 Field Documentation

5.4.3.1 btree_node_t* btree_t::root

Definition at line 18 of file btree.h.

5.4.3.2 uint32_t btree_t::size

Definition at line 19 of file btree.h.

The documentation for this struct was generated from the following file:

- include/structs/btree.h

5.5 cpu_cache Struct Reference

`#include <processor.h>`

Public Member Functions

- struct {
  unsigned line_size:12
  unsigned phy_line:10
  unsigned assoc:10
  } __attribute__ ((packed)) lpw
- struct {
  unsigned write_back:1
  unsigned inclusive:1
  unsigned indexing:1
  unsigned __pad0__:29
  } __attribute__ ((packed)) wcc

Data Fields

- uint32_t eax
- uint32_t sets

5.5.1 Detailed Description

Definition at line 135 of file processor.h.

5.5.2 Member Function Documentation

5.5.2.1 struct cpu_cache::@14 cpu_cache::__attribute__ ((packed))

5.5.2.2 struct cpu_cache::@15 cpu_cache::__attribute__ ((packed))

5.5.3 Field Documentation

5.5.3.1 unsigned cpu_cache::__pad0__

Definition at line 147 of file processor.h.

5.5.3.2 unsigned cpu_cache::assoc

Definition at line 140 of file processor.h.

5.5.3.3 uint32_t cpu_cache::eax

Definition at line 136 of file processor.h.

5.5.3.4 unsigned cpu_cache::inclusive

Definition at line 145 of file processor.h.

5.5.3.5 unsigned cpu_cache::indexing

Definition at line 146 of file processor.h.

5.5.3.6 unsigned cpu_cache::line_size

Definition at line 138 of file processor.h.

5.5.3.7 unsigned cpu_cache::phy_line

Definition at line 139 of file processor.h.

5.5.3.8 uint32_t cpu_cache::sets

Definition at line 142 of file processor.h.

5.5.3.9 unsigned cpu_cache::write_back

Definition at line 144 of file processor.h.

The documentation for this struct was generated from the following file:

- include/arch/x86/processor.h

5.6 cpu_state_t Struct Reference

Interaupt/Trap cpu state.

`#include <cpu_state.h>`

Data Fields

- reg_t ebp_frame
- reg_t eip_frame
- reg_t gs
- reg_t fs
- reg_t es
- reg_t ds
- reg_t edi
- reg_t esi
- reg_t o_ebp
- reg_t o_esp
- reg_t ebx
- reg_t edx
- reg_t ecx
- reg_t eax
- uint32_t error_code
- reg_t eip
- reg_t cs
- reg_t eflags
- reg_t esp
- reg_t ss

5.6.1 Detailed Description

Interaupt/Trap cpu state.

the cpu_state structure extends the default intel hardware interrupt/trap frame stored into stack. More detailed than the hardware frame.

Definition at line 25 of file cpu_state.h.

5.6.2 Field Documentation

5.6.2.1 reg_t cpu_state_t::cs

Definition at line 61 of file cpu_state.h.

5.6.2.2 reg_t cpu_state_t::ds

Definition at line 38 of file cpu_state.h.

5.6.2.3 reg_t cpu_state_t::eax

Definition at line 51 of file cpu_state.h.

5.6.2.4 reg_t cpu_state_t::ebp_frame

Definition at line 30 of file cpu_state.h.

5.6.2.5 reg_t cpu_state_t::ebx

Definition at line 48 of file cpu_state.h.

5.6.2.6 reg_t cpu_state_t::ecx

Definition at line 50 of file cpu_state.h.

5.6.2.7 reg_t cpu_state_t::edi

Definition at line 44 of file cpu_state.h.

5.6.2.8 reg_t cpu_state_t::edx

Definition at line 49 of file cpu_state.h.

5.6.2.9 reg_t cpu_state_t::eflags

Definition at line 62 of file cpu_state.h.

5.6.2.10 reg_t cpu_state_t::eip

Definition at line 60 of file cpu_state.h.

5.6.2.11 reg_t cpu_state_t::eip_frame

Definition at line 31 of file cpu_state.h.

5.6.2.12 uint32_t cpu_state_t::error_code

Definition at line 59 of file cpu_state.h.

5.6.2.13 reg_t cpu_state_t::es

Definition at line 37 of file cpu_state.h.

5.6.2.14 reg_t cpu_state_t::esi

Definition at line 45 of file cpu_state.h.

5.6.2.15 reg_t cpu_state_t::esp

Definition at line 63 of file cpu_state.h.

5.6.2.16 reg_t cpu_state_t::fs

Definition at line 36 of file cpu_state.h.

5.6.2.17 reg_t cpu_state_t::gs

Definition at line 35 of file cpu_state.h.

5.6.2.18 reg_t cpu_state_t::o_ebp

Definition at line 46 of file cpu_state.h.

5.6.2.19 reg_t cpu_state_t::o_esp

Definition at line 47 of file cpu_state.h.

5.6.2.20 reg_t cpu_state_t::ss

Definition at line 64 of file cpu_state.h.

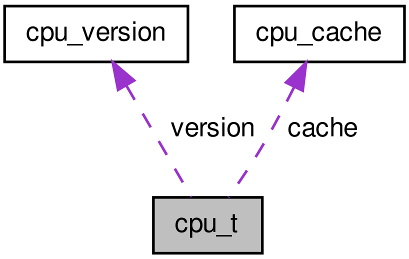
The documentation for this struct was generated from the following file:

- include/arch/x86/cpu_state.h

5.7 cpu_t Struct Reference

```
#include <processor.h>
```

Collaboration diagram for cpu_t:



Data Fields

- uint32_t vendor
- struct cpu_version version
- struct cpu_cache cache

5.7.1 Detailed Description

Definition at line 153 of file processor.h.

5.7.2 Field Documentation

5.7.2.1 struct cpu_cache cpu_t::cache

Definition at line 156 of file processor.h.

5.7.2.2 uint32_t cpu_t::vendor

Definition at line 154 of file processor.h.

5.7.2.3 struct cpu_version cpu_t::version

Definition at line 155 of file processor.h.

The documentation for this struct was generated from the following file:

- include/arch/x86/processor.h

5.8 cpu_version Struct Reference

```
#include <processor.h>
```

Public Member Functions

- struct {
  unsigned sse3:1
  unsigned __pad0__:4
  unsigned vmx:1
  unsigned smx:1
  unsigned est:1
  unsigned tm2:1
  unsigned ssse3:1
  unsigned __pad1__:7
  unsigned pcid:1
  unsigned dca:1
  unsigned sse4_1:1
  unsigned sse4_2:1
  unsigned x2apic:1
  unsigned __pad2__:10
  } __attribute__ ((packed)) featured_ecx

- struct {
  unsigned fpu:1
  unsigned vme:1
  unsigned de:1
  unsigned pse:1
  unsigned __pad0__:2
  unsigned pae:1
  unsigned __pad1__:2
  unsigned apic:1
  unsigned __pad2__:1
  unsigned sep:1
  unsigned pge:1
  unsigned mca:1
  unsigned cmov:1
  unsigned pat:1
  unsigned __pad3__:5
  unsigned acpi:1
  unsigned mmx:1
  unsigned fxsr:1
  unsigned sse:1
  unsigned sse2:1
  unsigned ss:1
  unsigned htt:1
  unsigned tm:1
  unsigned __pad4__:1

unsigned pbe:1
} __attribute__ ((packed)) featured_edx

Data Fields

- struct {
  unsigned stepping:4
  unsigned model:4
  unsigned family:4
  unsigned type:2
  unsigned ss:2
  unsigned exmodel:4
  unsigned exfamily:8
  unsigned sss:4
  } fms
- struct {
  char brand
  char cache
  char max_id
  char id
  } generic

5.8.1 Detailed Description

Definition at line 73 of file processor.h.

5.8.2 Member Function Documentation

5.8.2.1 struct cpu_version::@8 cpu_version::__attribute__ ((packed))

5.8.2.2 struct cpu_version::@9 cpu_version::__attribute__ ((packed))

5.8.3 Field Documentation

5.8.3.1 unsigned cpu_version::__pad0__

Definition at line 92 of file processor.h.

5.8.3.2 unsigned cpu_version::__pad1__

Definition at line 98 of file processor.h.

5.8.3.3 unsigned cpu_version::__pad2__

Definition at line 104 of file processor.h.

5.8.3.4 unsigned cpu_version::__pad3__

Definition at line 121 of file processor.h.

5.8.3.5 unsigned cpu_version::__pad4__

Definition at line 130 of file processor.h.

5.8.3.6 unsigned cpu_version::acpi

Definition at line 122 of file processor.h.

5.8.3.7 unsigned cpu_version::apic

Definition at line 114 of file processor.h.

5.8.3.8 char cpu_version::brand

Definition at line 85 of file processor.h.

5.8.3.9 char cpu_version::cache

Definition at line 86 of file processor.h.

5.8.3.10 unsigned cpu_version::cmov

Definition at line 119 of file processor.h.

5.8.3.11 unsigned cpu_version::dca

Definition at line 100 of file processor.h.

5.8.3.12 unsigned cpu_version::de

Definition at line 109 of file processor.h.

5.8.3.13 unsigned cpu_version::est

Definition at line 95 of file processor.h.

5.8.3.14 unsigned cpu_version::exfamily

Definition at line 81 of file processor.h.

5.8.3.15 unsigned cpu_version::exmodel

Definition at line 80 of file processor.h.

5.8.3.16 unsigned cpu_version::family

Definition at line 77 of file processor.h.

5.8.3.17 struct { ... } cpu_version::fms

5.8.3.18 unsigned cpu_version::fpu

Definition at line 107 of file processor.h.

5.8.3.19 unsigned cpu_version::fxsr

Definition at line 124 of file processor.h.

5.8.3.20 struct { ... } cpu_version::generic

5.8.3.21 unsigned cpu_version::htt

Definition at line 128 of file processor.h.

5.8.3.22 char cpu_version::id

Definition at line 88 of file processor.h.

5.8.3.23 char cpu_version::max_id

Definition at line 87 of file processor.h.

5.8.3.24 unsigned cpu_version::mca

Definition at line 118 of file processor.h.

5.8.3.25 unsigned cpu_version::mmx

Definition at line 123 of file processor.h.

5.8.3.26 unsigned cpu_version::model

Definition at line 76 of file processor.h.

5.8.3.27 unsigned cpu_version::pae

Definition at line 112 of file processor.h.

5.8.3.28 unsigned cpu_version::pat

Definition at line 120 of file processor.h.

5.8.3.29 unsigned cpu_version::pbe

Definition at line 131 of file processor.h.

5.8.3.30 unsigned cpu_version::pcid

Definition at line 99 of file processor.h.

5.8.3.31 unsigned cpu_version::pge

Definition at line 117 of file processor.h.

5.8.3.32 unsigned cpu_version::pse

Definition at line 110 of file processor.h.

5.8.3.33 unsigned cpu_version::sep

Definition at line 116 of file processor.h.

5.8.3.34 unsigned cpu_version::smx

Definition at line 94 of file processor.h.

5.8.3.35 unsigned cpu_version::ss

Definition at line 79 of file processor.h.

5.8.3.36 unsigned cpu_version::sse

Definition at line 125 of file processor.h.

5.8.3.37 unsigned cpu_version::sse2

Definition at line 126 of file processor.h.

5.8.3.38 unsigned cpu_version::sse3

Definition at line 91 of file processor.h.

5.8.3.39 unsigned cpu_version::sse4_1

Definition at line 101 of file processor.h.

5.8.3.40 unsigned cpu_version::sse4_2

Definition at line 102 of file processor.h.

5.8.3.41 unsigned cpu_version::sss

Definition at line 82 of file processor.h.

5.8.3.42 unsigned cpu_version::ssse3

Definition at line 97 of file processor.h.

5.8.3.43 unsigned cpu_version::stepping

Definition at line 75 of file processor.h.

5.8.3.44 unsigned cpu_version::tm

Definition at line 129 of file processor.h.

5.8.3.45 unsigned cpu_version::tm2

Definition at line 96 of file processor.h.

5.8.3.46 unsigned cpu_version::type

Definition at line 78 of file processor.h.

5.8.3.47 unsigned cpu_version::vme

Definition at line 108 of file processor.h.

5.8.3.48 unsigned cpu_version::vmx

Definition at line 93 of file processor.h.

5.8.3.49 unsigned cpu_version::x2apic

Definition at line 103 of file processor.h.

The documentation for this struct was generated from the following file:

- include/arch/x86/processor.h

5.9 cpuid_regs Struct Reference

Data Fields

- uint32_t eax
- uint32_t ebx
- uint32_t ecx
- uint32_t edx

5.9.1 Detailed Description

Definition at line 5 of file cpuid.c.

5.9.2 Field Documentation

5.9.2.1 uint32_t cpuid_regs::eax

Definition at line 6 of file cpuid.c.

5.9.2.2 uint32_t cpuid_regs::ebx

Definition at line 7 of file cpuid.c.

5.9.2.3 uint32_t cpuid_regs::ecx

Definition at line 8 of file cpuid.c.

5.9.2.4 uint32_t cpuid_regs::edx

Definition at line 9 of file cpuid.c.

The documentation for this struct was generated from the following file:

- kernel/generic/cpuid.c

5.10 cpuid_t Struct Reference

```
#include <cpuid.h>
```

Data Fields

- uint32_t eax
- uint32_t ebx
- uint32_t ecx
- uint32_t edx

5.10.1 Detailed Description

Definition at line 18 of file cpuid.h.

5.10.2 Field Documentation

5.10.2.1 uint32_t cpuid_t::eax

Definition at line 20 of file cpuid.h.

5.10.2.2 uint32_t cpuid_t::ebx

Definition at line 21 of file cpuid.h.

5.10.2.3 uint32_t cpuid_t::ecx

Definition at line 22 of file cpuid.h.

5.10.2.4 uint32_t cpuid_t::edx

Definition at line 23 of file cpuid.h.

The documentation for this struct was generated from the following file:

- include/cpuid.h

5.11 direntry_t Struct Reference

```
#include <vfs.h>
```

Data Fields

- char name [128]
- uint32_t inode_n

5.11.1 Detailed Description

Definition at line 29 of file vfs.h.

5.11.2 Field Documentation

5.11.2.1 uint32_t direntry_t::inode_n

Definition at line 31 of file vfs.h.

5.11.2.2 char direntry_t::name[128]

Definition at line 30 of file vfs.h.

The documentation for this struct was generated from the following file:

- kernel/vfs/vfs.h

5.12 element Struct Reference

```
#include <list.h>
```

5.12.1 Detailed Description

Definition at line 4 of file list.h.

The documentation for this struct was generated from the following file:

- include/structs/list.h

5.13 elfhdr Struct Reference

Elf binary header values.

`#include <elf.h>`

Data Fields

- uint32_t magic
- uint8_t magic2 [MAGIC_LEN-4]
- uint16_t type
- uint16_t machine
- uint32_t version
- uint32_t entry
- uint32_t phroff
- uint32_t shroff
- uint32_t flags
- uint16_t ehsize
- uint16_t phrsize
- uint16_t phrnum
- uint16_t shrsize
- uint16_t shrnum
- uint16_t shrstrtbl
- uint8_t magic [MAGIC_LEN]

5.13.1 Detailed Description

Elf binary header values.

Definition at line 49 of file elf.h.

5.13.2 Field Documentation

5.13.2.1 uint16_t elfhdr::ehsize

Elf header size

Definition at line 59 of file elf.h.

5.13.2.2 uint32_t elfhdr::entry

Address of entry point

Definition at line 55 of file elf.h.

5.13.2.3 uint32_t elfhdr::flags

Flags

Definition at line 58 of file elf.h.

5.13.2.4 uint16_t elfhdr::machine

Machine type

Definition at line 53 of file elf.h.

5.13.2.5 uint32_t elfhdr::magic

Definition at line 50 of file elf.h.

5.13.2.6 uint8_t elfhdr::magic[MAGIC_LEN]

the elf magic number

Definition at line 90 of file elf.h.

5.13.2.7 uint8_t elfhdr::magic2[MAGIC_LEN-4]

Definition at line 51 of file elf.h.

5.13.2.8 uint16_t elfhdr::phrnum

Number of program headers

Definition at line 61 of file elf.h.

5.13.2.9 uint32_t elfhdr::phroff

offset of program headers

Definition at line 56 of file elf.h.

5.13.2.10 uint16_t elfhdr::phrsize

Program header size

Definition at line 60 of file elf.h.

5.13.2.11 uint16_t elfhdr::shrnum

Number of section headers

Definition at line 63 of file elf.h.

5.13.2.12 uint32_t elfhdr::shroff

offset of Section headers

Definition at line 57 of file elf.h.

5.13.2.13 uint16_t elfhdr::shrsize

Section header size

Definition at line 62 of file elf.h.

5.13.2.14 uint16_t elfhdr::shrstrtbl

Section header string table

Definition at line 64 of file elf.h.

5.13.2.15 uint16_t elfhdr::type

File type (EXEC/RELOC..etc)

Definition at line 52 of file elf.h.

5.13.2.16 uint32_t elfhdr::version

Version

Definition at line 54 of file elf.h.

The documentation for this struct was generated from the following files:

- include/arch/x86/elf.h
- arch/x86/include/elf.h

5.14 ext2_dir_entry Struct Reference

```
#include <ext2fs.h>
```

Data Fields

- uint32_t inode
- uint16_t rec_len
- uint8_t name_len
- uint8_t type int8_t name [EXT2_NAME_LEN]

5.14.1 Detailed Description

Definition at line 172 of file ext2fs.h.

5.14.2 Field Documentation

5.14.2.1 uint32_t ext2_dir_entry::inode

Definition at line 173 of file ext2fs.h.

5.14.2.2 uint8_t type int8_t ext2_dir_entry::name[EXT2_NAME_LEN]

Definition at line 180 of file ext2fs.h.

5.14.2.3 uint8_t ext2_dir_entry::name_len

Definition at line 175 of file ext2fs.h.

5.14.2.4 uint16_t ext2_dir_entry::rec_len

Definition at line 174 of file ext2fs.h.

The documentation for this struct was generated from the following file:

- include/fs/ext2fs.h

5.15 ext2_group_desc Struct Reference

```
#include <ext2fs.h>
```

Data Fields

- uint32_t bg_block_bitmap
- uint32_t bg_inode_bitmap
- uint32_t bg_inode_table
- uint16_t bg_free_blocks_count
- uint16_t bg_free_inodes_count
- uint16_t bg_used_dirs_count
- uint16_t bg_pad
- uint32_t bg_reserved [3]

5.15.1 Detailed Description

Definition at line 184 of file ext2fs.h.

5.15.2 Field Documentation

5.15.2.1 uint32_t ext2_group_desc::bg_block_bitmap

Definition at line 186 of file ext2fs.h.

5.15.2.2 uint16_t ext2_group_desc::bg_free_blocks_count

Definition at line 189 of file ext2fs.h.

5.15.2.3 uint16_t ext2_group_desc::bg_free_inodes_count

Definition at line 190 of file ext2fs.h.

5.15.2.4 uint32_t ext2_group_desc::bg_inode_bitmap

Definition at line 187 of file ext2fs.h.

5.15.2.5 uint32_t ext2_group_desc::bg_inode_table

Definition at line 188 of file ext2fs.h.

5.15.2.6 uint16_t ext2_group_desc::bg_pad

Definition at line 192 of file ext2fs.h.

5.15.2.7 uint32_t ext2_group_desc::bg_reserved[3]

Definition at line 193 of file ext2fs.h.

5.15.2.8 uint16_t ext2_group_desc::bg_used_dirs_count

Definition at line 191 of file ext2fs.h.

The documentation for this struct was generated from the following file:

- include/fs/ext2fs.h

5.16 ext2_inode Struct Reference

```
#include <ext2fs.h>
```

Data Fields

- uint16_t i_mode
- uint16_t i_uid
- uint32_t i_size
- uint32_t i_atime
- uint32_t i_ctime
- uint32_t i_mtime
- uint32_t i_dtime
- uint16_t i_gid
- uint16_t i_links_count
- uint32_t i_blocks
- uint32_t i_flags
- uint32_t i_reserved1
- uint32_t i_block [EXT2_N_BLOCKS]
- uint32_t i_version
- uint32_t i_file_acl
- uint32_t i_dir_acl
- uint32_t i_faddr
- union {
    struct {
      uint8_t l_i_frag
      uint8_t l_i_fsize
      uint16_t l_i_reserved1
      uint16_t l_i_uid_high
      uint16_t l_i_gid_high
      uint32_t l_i_reserved2
    } linux
    struct {

uint8_t h_i_frag
uint8_t h_i_fsize
uint16_t h_i_mode_high
uint16_t h_i_uid_high
uint16_t h_i_gid_high
uint32_t h_i_author
} hurd2
struct {
uint8_t m_i_frag
uint8_t m_i_fsize
uint16_t m_pad1
uint32_t m_i_reserved2 [2]
} masix2
} osd2

5.16.1 Detailed Description

Definition at line 197 of file ext2fs.h.

5.16.2 Field Documentation

5.16.2.1 uint32_t ext2_inode::h_i_author

Definition at line 231 of file ext2fs.h.

5.16.2.2 uint8_t ext2_inode::h_i_frag

Definition at line 226 of file ext2fs.h.

5.16.2.3 uint8_t ext2_inode::h_i_fsize

Definition at line 227 of file ext2fs.h.

5.16.2.4 uint16_t ext2_inode::h_i_gid_high

Definition at line 230 of file ext2fs.h.

5.16.2.5 uint16_t ext2_inode::h_i_mode_high

Definition at line 228 of file ext2fs.h.

5.16.2.6 uint16_t ext2_inode::h_i_uid_high

Definition at line 229 of file ext2fs.h.

5.16.2.7 struct { ... } ext2_inode::hurd2

5.16.2.8 uint32_t ext2_inode::i_atime

Definition at line 201 of file ext2fs.h.

5.16.2.9 uint32_t ext2_inode::i_block[EXT2_N_BLOCKS]

Definition at line 210 of file ext2fs.h.

5.16.2.10 uint32_t ext2_inode::i_blocks

Definition at line 207 of file ext2fs.h.

5.16.2.11 uint32_t ext2_inode::i_ctime

Definition at line 202 of file ext2fs.h.

5.16.2.12 uint32_t ext2_inode::i_dir_acl

Definition at line 213 of file ext2fs.h.

5.16.2.13 uint32_t ext2_inode::i_dtime

Definition at line 204 of file ext2fs.h.

5.16.2.14 uint32_t ext2_inode::i_faddr

Definition at line 214 of file ext2fs.h.

5.16.2.15 uint32_t ext2_inode::i_file_acl

Definition at line 212 of file ext2fs.h.

5.16.2.16 uint32_t ext2_inode::i_flags

Definition at line 208 of file ext2fs.h.

5.16.2.17 uint16_t ext2_inode::i_gid

Definition at line 205 of file ext2fs.h.

5.16.2.18 uint16_t ext2_inode::i_links_count

Definition at line 206 of file ext2fs.h.

5.16.2.19 uint16_t ext2_inode::i_mode

Definition at line 198 of file ext2fs.h.

5.16.2.20 uint32_t ext2_inode::i_mtime

Definition at line 203 of file ext2fs.h.

5.16.2.21 uint32_t ext2_inode::i_reserved1

Definition at line 209 of file ext2fs.h.

5.16.2.22 uint32_t ext2_inode::i_size

Definition at line 200 of file ext2fs.h.

5.16.2.23 uint16_t ext2_inode::i_uid

Definition at line 199 of file ext2fs.h.

5.16.2.24 uint32_t ext2_inode::i_version

Definition at line 211 of file ext2fs.h.

5.16.2.25 uint8_t ext2_inode::l_i_frag

Definition at line 218 of file ext2fs.h.

5.16.2.26 uint8_t ext2_inode::l_i_fsize

Definition at line 219 of file ext2fs.h.

5.16.2.27 uint16_t ext2_inode::l_i_gid_high

Definition at line 222 of file ext2fs.h.

5.16.2.28 uint16_t ext2_inode::l_i_reserved1

Definition at line 220 of file ext2fs.h.

5.16.2.29 uint32_t ext2_inode::l_i_reserved2

Definition at line 223 of file ext2fs.h.

5.16.2.30 uint16_t ext2_inode::l_i_uid_high

Definition at line 221 of file ext2fs.h.

5.16.2.31 struct { ... } ext2_inode::linux

5.16.2.32 uint8_t ext2_inode::m_i_frag

Definition at line 234 of file ext2fs.h.

5.16.2.33 uint8_t ext2_inode::m_i_fsize

Definition at line 235 of file ext2fs.h.

5.16.2.34 uint32_t ext2_inode::m_i_reserved2[2]

Definition at line 237 of file ext2fs.h.

5.16.2.35 uint16_t ext2_inode::m_pad1

Definition at line 236 of file ext2fs.h.

5.16.2.36 struct { ... } ext2_inode::masix2

5.16.2.37 union { ... } ext2_inode::osd2

OS SPECIFIC VALUES from minix inode.h

The documentation for this struct was generated from the following file:

- include/fs/ext2fs.h

5.17 ext2_super_block Struct Reference

```
#include <ext2fs.h>
```

Data Fields

- uint32_t s_inodes_count
- uint32_t s_blocks_count
- uint32_t s_r_blocks_count
- uint32_t s_free_blocks_count
- uint32_t s_free_inodes_count
- uint32_t s_first_data_block
- uint32_t s_log_block_size
- uint32_t s_log_frag_size
- uint32_t s_blocks_per_group
- uint32_t s_frags_per_group
- uint32_t s_inodes_per_group
- uint32_t s_mtime
- uint32_t s_wtime
- uint16_t s_mnt_count
- uint16_t s_max_mnt_count
- uint16_t s_magic
- uint16_t s_state
- uint16_t s_errors
- uint16_t s_min_pad
- uint32_t s_lastcheck
- uint32_t s_checkinterval
- uint32_t s_os_id
- uint32_t s_maj_pad
- uint16_t s_uid
- uint16_t s_gid
- uint32_t s_first_ino
- uint16_t s_inode_size
- uint16_t s_block_group_nr
- uint32_t s_feature_compat
- uint32_t s_feature_incompat
- uint32_t s_feature_ro_compat
- uint8_t s_uuid [16]
- char s_volume_name [16]
- char s_last_mounted [64]
- uint32_t s_algorithm_usage_bitmap
- uint8_t s_prealloc_blocks
- uint8_t s_prealloc_dir_blocks
- uint16_t s_padding1
- uint8_t s_journal_uuid [16]
- uint32_t s_journal_inum

- uint32_t s_journal_dev
- uint32_t s_last_orphan
- uint32_t s_reserved [197]

5.17.1 Detailed Description

Definition at line 122 of file ext2fs.h.

5.17.2 Field Documentation

5.17.2.1 uint32_t ext2_super_block::s_algorithm_usage_bitmap

Definition at line 159 of file ext2fs.h.

5.17.2.2 uint16_t ext2_super_block::s_block_group_nr

Definition at line 152 of file ext2fs.h.

5.17.2.3 uint32_t ext2_super_block::s_blocks_count

Definition at line 124 of file ext2fs.h.

5.17.2.4 uint32_t ext2_super_block::s_blocks_per_group

Definition at line 131 of file ext2fs.h.

5.17.2.5 uint32_t ext2_super_block::s_checkinterval

Definition at line 143 of file ext2fs.h.

5.17.2.6 uint16_t ext2_super_block::s_errors

Definition at line 140 of file ext2fs.h.

5.17.2.7 uint32_t ext2_super_block::s_feature_compat

Definition at line 153 of file ext2fs.h.

5.17.2.8 uint32_t ext2_super_block::s_feature_incompat

Definition at line 154 of file ext2fs.h.

5.17.2.9 uint32_t ext2_super_block::s_feature_ro_compat

Definition at line 155 of file ext2fs.h.

5.17.2.10 uint32_t ext2_super_block::s_first_data_block

Definition at line 128 of file ext2fs.h.

5.17.2.11 uint32_t ext2_super_block::s_first_ino

Definition at line 150 of file ext2fs.h.

5.17.2.12 uint32_t ext2_super_block::s_frags_per_group

Definition at line 132 of file ext2fs.h.

5.17.2.13 uint32_t ext2_super_block::s_free_blocks_count

Definition at line 126 of file ext2fs.h.

5.17.2.14 uint32_t ext2_super_block::s_free_inodes_count

Definition at line 127 of file ext2fs.h.

5.17.2.15 uint16_t ext2_super_block::s_gid

Definition at line 147 of file ext2fs.h.

5.17.2.16 uint16_t ext2_super_block::s_inode_size

Definition at line 151 of file ext2fs.h.

5.17.2.17 uint32_t ext2_super_block::s_inodes_count

Definition at line 123 of file ext2fs.h.

5.17.2.18 uint32_t ext2_super_block::s_inodes_per_group

Definition at line 133 of file ext2fs.h.

5.17.2.19 uint32_t ext2_super_block::s_journal_dev

Definition at line 165 of file ext2fs.h.

5.17.2.20 uint32_t ext2_super_block::s_journal_inum

Definition at line 164 of file ext2fs.h.

5.17.2.21 uint8_t ext2_super_block::s_journal_uuid[16]

Definition at line 163 of file ext2fs.h.

5.17.2.22 char ext2_super_block::s_last_mounted[64]

Definition at line 158 of file ext2fs.h.

5.17.2.23 uint32_t ext2_super_block::s_last_orphan

Definition at line 166 of file ext2fs.h.

5.17.2.24 uint32_t ext2_super_block::s_lastcheck

Definition at line 142 of file ext2fs.h.

5.17.2.25 uint32_t ext2_super_block::s_log_block_size

Definition at line 129 of file ext2fs.h.

5.17.2.26 uint32_t ext2_super_block::s_log_frag_size

Definition at line 130 of file ext2fs.h.

5.17.2.27 uint16_t ext2_super_block::s_magic

Definition at line 138 of file ext2fs.h.

5.17.2.28 uint32_t ext2_super_block::s_maj_pad

Definition at line 145 of file ext2fs.h.

5.17.2.29 uint16_t ext2_super_block::s_max_mnt_count

Definition at line 137 of file ext2fs.h.

5.17.2.30 uint16_t ext2_super_block::s_min_pad

Definition at line 141 of file ext2fs.h.

5.17.2.31 uint16_t ext2_super_block::s_mnt_count

Definition at line 136 of file ext2fs.h.

5.17.2.32 uint32_t ext2_super_block::s_mtime

Definition at line 134 of file ext2fs.h.

5.17.2.33 uint32_t ext2_super_block::s_os_id

Definition at line 144 of file ext2fs.h.

5.17.2.34 uint16_t ext2_super_block::s_padding1

Definition at line 162 of file ext2fs.h.

5.17.2.35 uint8_t ext2_super_block::s_prealloc_blocks

Definition at line 160 of file ext2fs.h.

5.17.2.36 uint8_t ext2_super_block::s_prealloc_dir_blocks

Definition at line 161 of file ext2fs.h.

5.17.2.37 uint32_t ext2_super_block::s_r_blocks_count

Definition at line 125 of file ext2fs.h.

5.17.2.38 uint32_t ext2_super_block::s_reserved[197]

Definition at line 167 of file ext2fs.h.

5.17.2.39 uint16_t ext2_super_block::s_state

Definition at line 139 of file ext2fs.h.

5.17.2.40 uint16_t ext2_super_block::s_uid

Definition at line 146 of file ext2fs.h.

5.17.2.41 uint8_t ext2_super_block::s_uuid[16]

Definition at line 156 of file ext2fs.h.

5.17.2.42 char ext2_super_block::s_volume_name[16]

Definition at line 157 of file ext2fs.h.

5.17.2.43 uint32_t ext2_super_block::s_wtime

Definition at line 135 of file ext2fs.h.

The documentation for this struct was generated from the following file:

- include/fs/ext2fs.h

5.18 Gdtdesc Struct Reference

```
#include <memvals.h>
```

Data Fields

- uint16_t size
- uint32_t base

5.18.1 Detailed Description

Definition at line 46 of file memvals.h.

5.18.2 Field Documentation

5.18.2.1 uint32_t Gdtdesc::base

Definition at line 48 of file memvals.h.

5.18.2.2 uint16_t Gdtdesc::size

Definition at line 47 of file memvals.h.

The documentation for this struct was generated from the following file:

- include/memvals.h

5.19 gpr_regs_t Struct Reference

General purpose registers.

`#include <cpu_state.h>`

Data Fields

- reg_t edi
- reg_t esi
- reg_t ebp
- reg_t esp
- reg_t ebx
- reg_t edx
- reg_t ecx
- reg_t eax

5.19.1 Detailed Description

General purpose registers.

gpr registers structure contains general purpose register. ordered to be filled by a single PUSHAD/PUSHAL

Definition at line 74 of file cpu_state.h.

5.19.2 Field Documentation

5.19.2.1 reg_t gpr_regs_t::eax

Definition at line 83 of file cpu_state.h.

5.19.2.2 reg_t gpr_regs_t::ebp

Definition at line 78 of file cpu_state.h.

5.19.2.3 reg_t gpr_regs_t::ebx

Definition at line 80 of file cpu_state.h.

5.19.2.4 reg_t gpr_regs_t::ecx

Definition at line 82 of file cpu_state.h.

5.19.2.5 reg_t gpr_regs_t::edi

Definition at line 76 of file cpu_state.h.

5.19.2.6 reg_t gpr_regs_t::edx

Definition at line 81 of file cpu_state.h.

5.19.2.7 reg_t gpr_regs_t::esi

Definition at line 77 of file cpu_state.h.

5.19.2.8 reg_t gpr_regs_t::esp

Definition at line 79 of file cpu_state.h.

The documentation for this struct was generated from the following file:

- include/arch/x86/cpu_state.h

5.20 htable_node_t Struct Reference

```
#include <htable.h>
```

Data Fields

- uint32_t key
- uint32_t value

5.20.1 Detailed Description

Definition at line 24 of file htable.h.

5.20.2 Field Documentation

5.20.2.1 uint32_t htable_node_t::key

Definition at line 25 of file htable.h.

5.20.2.2 uint32_t htable_node_t::value

Definition at line 26 of file htable.h.

The documentation for this struct was generated from the following file:

- include/structs/htable.h

5.21 htable_t Struct Reference

a hash table structure

`#include <htable.h>`

Data Fields

- void ** table
- uint32_t buckets
- uint32_t * bucket_size
- uint32_t(* hash)(uint32_t)
- uint32_t(* destroy)(uint32_t)
- uint32_t size
- uint32_t bucket_length

5.21.1 Detailed Description

a hash table structure

Hash table holds a pointer-to-pointer which is considered the actual table, It also holds the bucket_size array which tracks the size of buckets the buckets which represents the number of buckets, a hash function passed on inialization time, same as destroy function. the overall size which is used to determine the overall inserted values count.

Definition at line 29 of file htable.h.

5.21.2 Field Documentation

5.21.2.1 uint32_t htable_t::bucket_length

Definition at line 36 of file htable.h.

5.21.2.2 uint32_t* htable_t::bucket_size

Definition at line 32 of file htable.h.

5.21.2.3 uint32_t htable_t::buckets

Definition at line 31 of file htable.h.

5.21.2.4 uint32_t(* htable_t::destroy)(uint32_t)

Definition at line 34 of file htable.h.

5.21.2.5 uint32_t(* htable_t::hash)(uint32_t)

Definition at line 33 of file htable.h.

5.21.2.6 uint32_t htable_t::size

Definition at line 35 of file htable.h.

5.21.2.7 void** htable_t::table

Definition at line 30 of file htable.h.

The documentation for this struct was generated from the following file:

- include/structs/htable.h

5.22 initrd_file_header_t Struct Reference

```
#include <initrd.h>
```

Data Fields

- uint8_t magic
- char name [64]
- uint32_t offset
- uint32_t size

5.22.1 Detailed Description

Definition at line 25 of file initrd.h.

5.22.2 Field Documentation

5.22.2.1 uint8_t initrd_file_header_t::magic

Definition at line 27 of file initrd.h.

5.22.2.2 char initrd_file_header_t::name[64]

Definition at line 28 of file initrd.h.

5.22.2.3 uint32_t initrd_file_header_t::offset

Definition at line 29 of file initrd.h.

5.22.2.4 uint32_t initrd_file_header_t::size

Definition at line 30 of file initrd.h.

The documentation for this struct was generated from the following file:

- kernel/vfs/initrd.h

5.23 initrd_header_t Struct Reference

```
#include <initrd.h>
```

Data Fields

- uint32_t nfiles

5.23.1 Detailed Description

Definition at line 20 of file initrd.h.

5.23.2 Field Documentation

5.23.2.1 uint32_t initrd_header_t::nfiles

Definition at line 22 of file initrd.h.

The documentation for this struct was generated from the following file:

- kernel/vfs/initrd.h

5.24 kcommand Struct Reference

`#include <kconsole.h>`

Data Fields

- const char ∗ name
- const char ∗ hint
- void(∗ operation)(void)

5.24.1 Detailed Description

Definition at line 17 of file kconsole.h.

5.24.2 Field Documentation

5.24.2.1 const char∗ kcommand::hint

Definition at line 19 of file kconsole.h.

5.24.2.2 const char∗ kcommand::name

Definition at line 18 of file kconsole.h.

5.24.2.3 void(∗ kcommand::operation)(void)

Definition at line 20 of file kconsole.h.

The documentation for this struct was generated from the following file:

- include/kconsole.h

5.25 Page Struct Reference

a page list entry, used to hold info about pages.

`#include <page.h>`

Data Fields

- page_entry_t link
- uint16_t ref

5.25.1 Detailed Description

a page list entry, used to hold info about pages.

Definition at line 199 of file page.h.

5.25.2 Field Documentation

5.25.2.1 page_entry_t Page::link

Definition at line 201 of file page.h.

5.25.2.2 uint16_t Page::ref

Definition at line 202 of file page.h.

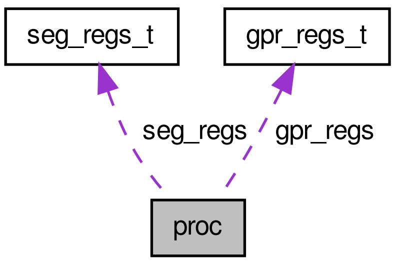
The documentation for this struct was generated from the following file:

- include/arch/x86/mm/page.h

5.26 proc Struct Reference

```
#include <proc.h>
```

Collaboration diagram for proc:



Public Member Functions

- LIST_ENTRY (proc) link
- LIFO_ENTRY (proc) q_link
- LIST_ENTRY (proc) wait_link

Data Fields

- gpr_regs_t gpr_regs
- seg_regs_t seg_regs
- reg_t eip
- uint32_t cs
- reg_t eflags
- reg_t esp
- uint32_t ss
- uint32_t id
- uint32_t status
- pde_t * page_directory
- uint32_t cr3
- uint32_t preempted
- uint32_t dequeqed
- uint32_t timer

5.26.1 Detailed Description

Definition at line 23 of file proc.h.

5.26.2 Member Function Documentation

5.26.2.1 proc::LIFO_ENTRY (proc)

5.26.2.2 proc::LIST_ENTRY (proc)

5.26.2.3 proc::LIST_ENTRY (proc)

5.26.3 Field Documentation

5.26.3.1 uint32_t proc::cr3

Definition at line 35 of file proc.h.

5.26.3.2 uint32_t proc::cs

Definition at line 27 of file proc.h.

5.26.3.3 uint32_t proc::dequeqed

Definition at line 37 of file proc.h.

5.26.3.4 reg_t proc::eflags

Definition at line 28 of file proc.h.

5.26.3.5 reg_t proc::eip

Definition at line 26 of file proc.h.

5.26.3.6 reg_t proc::esp

Definition at line 29 of file proc.h.

5.26.3.7 gpr_regs_t proc::gpr_regs

Definition at line 24 of file proc.h.

5.26.3.8 uint32_t proc::id

Definition at line 31 of file proc.h.

5.26.3.9 pde_t* proc::page_directory

Definition at line 34 of file proc.h.

5.26.3.10 uint32_t proc::preempted

Definition at line 36 of file proc.h.

5.26.3.11 seg_regs_t proc::seg_regs

Definition at line 25 of file proc.h.

5.26.3.12 uint32_t proc::ss

Definition at line 30 of file proc.h.

5.26.3.13 uint32_t proc::status

Definition at line 32 of file proc.h.

5.26.3.14 uint32_t proc::timer

Definition at line 43 of file proc.h.

The documentation for this struct was generated from the following file:

- include/proc/proc.h

5.27 proghdr Struct Reference

An ELF program header structure.

```
#include <elf.h>
```

Data Fields

- uint32_t type
- uint32_t offset
- uint32_t vaddr
- uint32_t paddr
- uint32_t filesz
- uint32_t memsz
- uint32_t flags
- uint32_t align

5.27.1 Detailed Description

An ELF program header structure.

Definition at line 71 of file elf.h.

5.27.2 Field Documentation

5.27.2.1 uint32_t proghdr::align

Definition at line 79 of file elf.h.

5.27.2.2 uint32_t proghdr::filesz

Definition at line 76 of file elf.h.

5.27.2.3 uint32_t proghdr::flags

Definition at line 78 of file elf.h.

5.27.2.4 uint32_t proghdr::memsz

Definition at line 77 of file elf.h.

5.27.2.5 uint32_t proghdr::offset

Definition at line 73 of file elf.h.

5.27.2.6 uint32_t proghdr::paddr

Definition at line 75 of file elf.h.

5.27.2.7 uint32_t proghdr::type

Definition at line 72 of file elf.h.

5.27.2.8 uint32_t proghdr::vaddr

Definition at line 74 of file elf.h.

The documentation for this struct was generated from the following files:

- include/arch/x86/elf.h
- arch/x86/include/elf.h

5.28 sechdr Struct Reference

An ELF section header structure.

```
#include <elf.h>
```

Data Fields

- uint32_t name
- uint32_t type
- uint32_t flags
- uint32_t addr
- uint32_t offset
- uint32_t size
- uint32_t link
- uint32_t info
- uint32_t addralign
- uint32_t entsize

5.28.1 Detailed Description

An ELF section header structure.

Definition at line 82 of file elf.h.

5.28.2 Field Documentation

5.28.2.1 uint32_t sechdr::addr

Definition at line 86 of file elf.h.

5.28.2.2 uint32_t sechdr::addralign

Definition at line 91 of file elf.h.

5.28.2.3 uint32_t sechdr::entsize

Definition at line 92 of file elf.h.

5.28.2.4 uint32_t sechdr::flags

Definition at line 85 of file elf.h.

5.28.2.5 uint32_t sechdr::info

Definition at line 90 of file elf.h.

5.28.2.6 uint32_t sechdr::link

Definition at line 89 of file elf.h.

5.28.2.7 uint32_t sechdr::name

Definition at line 83 of file elf.h.

5.28.2.8 uint32_t sechdr::offset

Definition at line 87 of file elf.h.

5.28.2.9 uint32_t sechdr::size

Definition at line 88 of file elf.h.

5.28.2.10 uint32_t sechdr::type

Definition at line 84 of file elf.h.

The documentation for this struct was generated from the following files:

- include/arch/x86/elf.h
- arch/x86/include/elf.h

5.29 seg_regs_t Struct Reference

Segment registers.

```
#include <cpu_state.h>
```

Data Fields

- reg_t gs
- reg_t fs
- reg_t es
- reg_t ds

5.29.1 Detailed Description

Segment registers.

the structure contains segment registers except for SP and CS since they are stored by the hardware into the structure.

Definition at line 93 of file cpu_state.h.

5.29.2 Field Documentation

5.29.2.1 reg_t seg_regs_t::ds

Definition at line 97 of file cpu_state.h.

5.29.2.2 reg_t seg_regs_t::es

Definition at line 96 of file cpu_state.h.

5.29.2.3 reg_t seg_regs_t::fs

Definition at line 95 of file cpu_state.h.

5.29.2.4 reg_t seg_regs_t::gs

Definition at line 94 of file cpu_state.h.

The documentation for this struct was generated from the following file:

- include/arch/x86/cpu_state.h

5.30 Segdesc Struct Reference

```
#include <memvals.h>
```

Data Fields

- unsigned limit_0: 16
- unsigned base_0: 16
- unsigned base_1: 8
- unsigned permission: 8
- unsigned limit_1: 4
- unsigned flags: 4
- unsigned base: 8

5.30.1 Detailed Description

Definition at line 34 of file memvals.h.

5.30.2 Field Documentation

5.30.2.1 unsigned Segdesc::base

Definition at line 41 of file memvals.h.

5.30.2.2 unsigned Segdesc::base_0

Definition at line 36 of file memvals.h.

5.30.2.3 unsigned Segdesc::base_1

Definition at line 37 of file memvals.h.

5.30.2.4 unsigned Segdesc::flags

Definition at line 40 of file memvals.h.

5.30.2.5 unsigned Segdesc::limit_0

Definition at line 35 of file memvals.h.

5.30.2.6 unsigned Segdesc::limit_1

Definition at line 39 of file memvals.h.

5.30.2.7 unsigned Segdesc::permission

Definition at line 38 of file memvals.h.

The documentation for this struct was generated from the following file:

- include/memvals.h

5.31 semaphore_t Struct Reference

```
#include <semaphore.h>
```

Data Fields

- uint32_t count
- struct Proc_List wait_list

5.31.1 Detailed Description

Definition at line 14 of file semaphore.h.

5.31.2 Field Documentation

5.31.2.1 uint32_t semaphore_t::count

Definition at line 15 of file semaphore.h.

5.31.2.2 struct Proc_List semaphore_t::wait_list

Definition at line 16 of file semaphore.h.

The documentation for this struct was generated from the following file:

- include/synchronization/semaphore.h

5.32 spinlock Struct Reference

```
#include <spinlock.h>
```

Data Fields

- uint32_t lock

5.32.1 Detailed Description

Definition at line 4 of file spinlock.h.

5.32.2 Field Documentation

5.32.2.1 uint32_t spinlock::lock

Definition at line 5 of file spinlock.h.

The documentation for this struct was generated from the following file:

- include/synchronization/spinlock.h

5.33 spinlock_t Struct Reference

```
#include <spinlock.h>
```

Data Fields

- atomic_t lock

5.33.1 Detailed Description

Definition at line 11 of file spinlock.h.

5.33.2 Field Documentation

5.33.2.1 atomic_t spinlock_t::lock

Definition at line 12 of file spinlock.h.

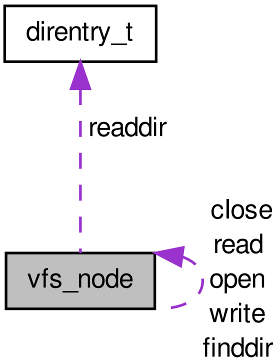
The documentation for this struct was generated from the following file:

- include/arch/x86/spinlock.h

5.34 vfs_node Struct Reference

```
#include <vfs.h>
```

Collaboration diagram for vfs_node:



Data Fields

- char name [128]
- uint32_t inode
- uint32_t mask
- uint32_t uid
- uint32_t gid
- uint32_t flags
- uint32_t size
- uint32_t impl
- read_type_t read
- write_type_t write
- open_type_t open
- close_type_t close
- readdir_type_t readdir
- finddir_type_t finddir
- struct fs_node * ptr

5.34.1 Detailed Description

Definition at line 37 of file vfs.h.

5.34.2 Field Documentation

5.34.2.1 close_type_t vfs_node::close

Definition at line 50 of file vfs.h.

5.34.2.2 finddir_type_t vfs_node::finddir

Definition at line 52 of file vfs.h.

5.34.2.3 uint32_t vfs_node::flags

Definition at line 44 of file vfs.h.

5.34.2.4 uint32_t vfs_node::gid

Definition at line 43 of file vfs.h.

5.34.2.5 uint32_t vfs_node::impl

Definition at line 46 of file vfs.h.

5.34.2.6 uint32_t vfs_node::inode

Definition at line 40 of file vfs.h.

5.34.2.7 uint32_t vfs_node::mask

Definition at line 41 of file vfs.h.

5.34.2.8 char vfs_node::name[128]

Definition at line 39 of file vfs.h.

5.34.2.9 open_type_t vfs_node::open

Definition at line 49 of file vfs.h.

5.34.2.10 struct fs_node* vfs_node::ptr

Definition at line 53 of file vfs.h.

5.34.2.11 read_type_t vfs_node::read

Definition at line 47 of file vfs.h.

5.34.2.12 readdir_type_t vfs_node::readdir

Definition at line 51 of file vfs.h.

5.34.2.13 uint32_t vfs_node::size

Definition at line 45 of file vfs.h.

5.34.2.14 uint32_t vfs_node::uid

Definition at line 42 of file vfs.h.

5.34.2.15 write_type_t vfs_node::write

Definition at line 48 of file vfs.h.

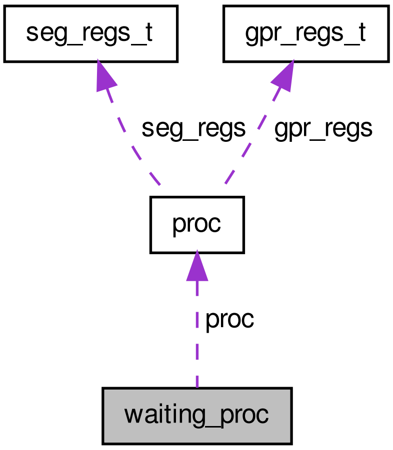
The documentation for this struct was generated from the following file:

- kernel/vfs/vfs.h

5.35 waiting_proc Struct Reference

```
#include <wait_queue.h>
```

Collaboration diagram for waiting_proc:



Public Member Functions

- LIST_ENTRY (waiting_proc) next

Data Fields

- proc_t ∗ proc

5.35.1 Detailed Description

Definition at line 13 of file wait_queue.h.

5.35.2 Member Function Documentation

5.35.2.1 waiting_proc::LIST_ENTRY (waiting_proc)

5.35.3 Field Documentation

5.35.3.1 proc_t∗ waiting_proc::proc

Definition at line 15 of file wait_queue.h.

The documentation for this struct was generated from the following file:

- include/synchronization/wait_queue.h