T.C.
Yıldız Technical University
Electric - Electronic Faculty

Computer Science & Engineering Department

StarTech®

Operating System Software

Junior Project

Project Supervizor

Prof. M.Yahya KARSLIGİL

The Project Team

9411009 Selçuk BAŞAK
9411011 Erdem HASEKİ
9411032 İrfan GÜNEYDAŞ

ÖNSÖZ

İşletim sistemi bilgisayar kullanıcıları ile bilgisayar donanımının arasında yer alan bir programdır. İşletim sistemlerinin amacı kullanıcılara programlarını rahat ve verimli olarak çalıştırabilecek bir ortam sağlamaktır. Bunun yanında bir işletim sistemi bilgisayar sistemi üzerinde yapılan işlemlerin doğruluğunu sağlamak zorundadır.

İlk bilgisayar sistemlerinde işletim sistemleri kullanılmıyordu. Yapılan işlemler doğrudan donanım üzerinden yapılıyordu. Bilgisayar teknolojisi geliştikçe, donanım daha da güçlendi ve bu işlemlerin elle yapılması olanaksızlaştı ve ilk olarak ilkel işletim sistemi benzeri yazılımlar kullanılmaya başlandı. Bu işletim sistemleri gelişerek günümüzün zaman paylaşımlı, çok görevli işletim sistemleri olmuşlardır.

Bizim temel amacımız modern bir işletim sistemi geliştirerek, bilgisayar bilimlerinin çok önemli bir konusu olan işletim sistemlerini daha yakından incelemek ve anlamak. Geliştirilecek sistemi daha önce yapılmış bir sistemden geliştirmek yerine en baştan tüm tasarımı kendimize ait bir sistem olmasını tercih ettik. Böylece yeni bir sistem tasarlanırken daha değişik sonuçlara ulaşabilme imkanımız oldu.

Proje Grubu
İstanbul,1997

PREFACE

An Operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. Beside that, the operating system must ensure correct operation of the computer system.

First computer system did not use an operating system. The operations were performed directly modifying hardware. As the computer technology advanced, computer systems got more power, these operations can not be performed directly on hardware system. As a result, first primitive operating systems was used. Those systems has turned into today's modern operating systems supporting time sharing, multitasking.

Our main aim is to analyze and understand operating systems which is a very important subject in computer science, by developing an operating system. Proposed system does not depend on any other operating system ,instead, we would rather develop a system of our own design. As a result, we may come across with some new aspects of operating system.

The Project Team
Istanbul,1997

SUMMARY

StarTech® is an operating system. It has based on PCs with an 80386 or better CPU. Its main attributes includes protection, multiprogramming, time slice based preemptive multitasking. It supports only one CPU PC systems. It uses protected mode to utilize protection. It uses a three layer architecture, applications runs at top and uses Application Program Interface(API) functions which is in the middle, and system kernel which runs at the lowest level. Kernel and API is divided into several parts so they can be thought as separate modules.

Writing an operating system software is a very advanced process. It requires a lot of research on both operating system concepts and computer system hardware. To some extend, StarTech® is designed to be modern system but it may contain some short comes, too.

ACKNOWLEDGMENTS

We appreciate any helps, advice, guidance about StarTech® and those who have shared their ideas, books, CDs, etc.

Special Thanks to:

Prof. M. Yahya Karslıgil
Yrd. Doç. Selim Akyokuş
Arş.Gör. Ahmet Haktanır
Arş.Gör. Ahmet Tevfik İnan

and also,*

Kerem Irgan
Onursal Tan
Sinan Kaya
Şeniz Gayde
Ülkü Gündüz

*(in the order of names)

CONTENTS

1. INTRODUCTİON

StarTech® is an operating system for PCs with 80386 and better CPUs. It is implemented as a protected-mode, stand-alone operating system that supports process based multitasking. StarTech® runs directly on the specified PC hardware without support from any other operating system. StarTech® implements virtual memory. Because of its 32 bit architecture, it is powerful. There will be a handful of demonstrations that illustrate the multi-programming and concurrency control and other subsystems.

2. SYSTEM ANALYSİS

An operating system is an interface between user programs and bare hardware. It should be easy to be used for users and easy to be implemented for system programmers. There are several approaches in operating system design but the system should use a modern one which is as possible as up to date system with its attributes. Today's desktop computers has even more power than old and huge mainframe systems. Desktop computers also known as personal computers (PC). PCs operating system has gained the features of those operating systems used on mainframes.

From the view of a user operating systems are just like servant that does what the user tells to do. Because of that, most operating systems redesigned their user interface and most of the popular operating systems uses a graphical user interface. But unfortunately there is no standard both easy to use and efficient. Actually, this is a major subject and may be a project on its own. As a result, a period of a term is not enough for developing a graphical user interface but it is assumed that it will have this feature in future versions.

IBM PC/AT compatible machines have a set of properties common, but it is generally supported by BIOS. 80386 and better systems have protected mode features but unfortunately BIOS is written in real mode and calling it through protected mode causes very high overhead(switch to V86 mode etc.). So, system device drivers should be written from the scratch.

Memory management using virtual memory is another good feature of 80386 and better processors. It should be used in a modern operating system which supports multiprogramming because there may not be enough physical memory for all process running at the same time.

At Last, StarTech® ,as a modern OS , will have these features listed below.

System kernel is simple ,containing only code directly interface with hardware, all other functions in the API. These are ,processor management, memory management, I/O system and device drivers and synchronization system. Processor management will use a time slice based preemptive method to implement multiprogramming. Memory management will use virtual memory. Device drivers are written only for those critical for the system ,that are keyboard, screen, floppy disks, hard disks, printers and communication ports.

API is not complex that is application programmers will surely welcome. These functions are interface between application programs and computer hardware or operating system.

User interface is on text screen for now. But it is expected that in version it will be graphical.

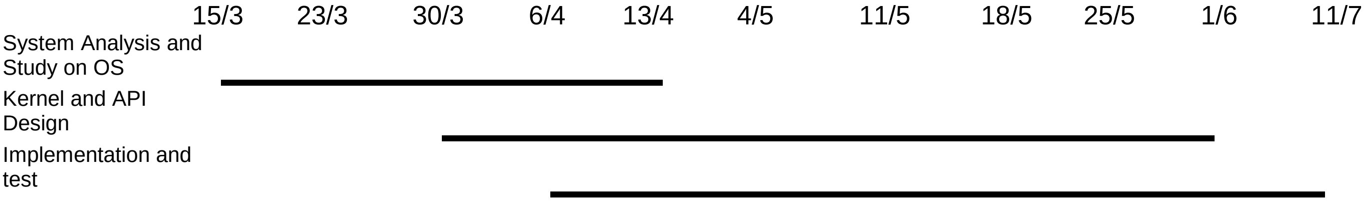
3. FEASİBİLİTY

Writing an operating system requires many resources and researches. Because StarTech® supports many modern features, many problems arises. These problems exists nearly all phases of the project. Designing is a really hard work but can be worked out. Testing will be really awkward. To develop the system there must be compiler for use.

BCC 3.1 as 16 bit C/C++ Compiler
BCC32 as 32 bit C/C++ Compiler.
WATCOM 10.0a C Compiler.
TASM32 as 80386 protected mode Assembler.
TLINK32 as Linker.

Another point is to develop compilers for StarTech® .Because of not having opportunity to develop a compiler , it is planned that DOS compilers producing 32 bit code with flat memory model will be used. The DOS executable program then post-compiled by a routine, written by us, which will then be able to run in StarTech®.

Project Timing:



Project Assignments:

Selçuk Başak	Process Management Synchronization Operations
Erdem Haseki	Main Memory Management File System Management
İrfan Güneydaş	I/O System Management System Command Interpreter Program, and some utility programs

4. SYSTEM DESİGN

The Architecture of StarTech®

<table>
<tr><td colspan="2">Command Interpreter</td><td colspan="2">Application Programs</td><td colspan="2">Compilers Utility Programs</td></tr>
<tr><td colspan="6">Application Program Interface</td></tr>
<tr><td colspan="3">Memory Management System</td><td colspan="3">File Management System</td></tr>
<tr><td colspan="2">I/O System & Device Management</td><td colspan="2">Process Management</td><td colspan="2">Synchronization System</td></tr>
<tr><td colspan="6">HARDWARE SYSTEM</td></tr>
</table>

General Descriptions of Subsystems

Process Management

- Keep track of processor and status of processes.
- Decide which process gets the processor, when, and how much (processor scheduler).
- Allocate the processor to a processor to a process by setting up necessary hardware registers.
- Reclaim processor when process relinquishes processor usage, terminates, or exceed allowed time of usage.
- Optimize the use of CPU time.

Synchronization System

- Keep track of the system resources that is to be shared among the processes.
- if necessary, it assures that a resource is allocated to a process and it will not be reallocated to another process at the same time.
- Handles the use limited system resources.

I/O System & Device Management

- Keep track of the devices, such as console, communication ,disk etc.
- Allow access to hardware using an abstraction.
- Responsible for choosing an efficient the way of low level data transfers between memory and external data storage or process devices.
- Allocation of the device and initiation of the I/O transfer.
- Optimize the use of devices.

Memory Management System

- Keep track of the memory. What parts are in use by which program and what parts are free.
- Decide which process gets the memory and how much.
- Allocate memory for a process when requested.
- Reclaim memory for later use.
- Optimize the use of memory.
- Prevents the user program from destroy operating system code and data .

File Management System

- Keep track of the data on the disks.
- Decide which process get use of the files.
- Allow process to access to disk files in an easy, fast and efficient way.
- Design of the actual physical device storage method.
- Implements accessing routines to files.

Application Program Interface

- Allows application programs to interact with operating system
- Abstracts the hardware devices for applications
- supports high level languages like C/C++.

Command Interpreter

- Interpret the user commands and apply them.
- Load application programs and initiate them.
- Implements many file operation that can be used from the command line.

Functional System Structure of StarTech®

<table>
<tr><td colspan="5">Application Programs</td></tr>
<tr><td colspan="5">Application Program Interface
(API)</td></tr>
<tr><td>Device
Drivers</td><td>Memory
Management
System</td><td>File
Management
System</td><td>Processor
Scheduler</td><td>Synchronization
System</td></tr>
<tr><td colspan="5">HARDWARE
SYSTEM</td></tr>
</table>

4.1. DISK BOOT-STRAP ROUTINE

Disk Boot-strapping process is the first of the three stage of loading the StarTech®. Disk Boot-strap routine is located at track 0,head 0,sector 1 of a floppy disk or the first sector in a harddisk partition. This routine can be at most 512 bytes. As a result it cannot load and initialize system. Its primary task is to load “System Initialization Routine” of the StartTech®(Part 2) by using bios interrupt 13h at 9000:0000h and to give control to it.

When an PC/AT machine is powered up or reset, control is transferred to 0FFFF:0h by the 80x86 CPU. At that location ROM-BIOS resides. It first test the system. This test is called POST(Power On Self Test) and will only occur by power on or cold reset(by pressing reset button). Then ROM-BIOS executes an “int 19h”. “int 19h” ,in return, attempts to load the sector at head 0, cylinder 0, sector 1, of a diskette or fixed disk into memory at 0:7C00h, The BIOS checks the sector to see if it has a boot signature (the value 055aah in the last two bytes of the sector). If the sector does have that signature, transfer control there. That is, CS is set to 0 and IP is set to 7C00h. This sector has the operating system bootstrap loader. At this point the processor is in 16-bit ``real'' mode, which still uses the Intel segmented architecture. The entire boot sector is written in assembly code.

Memory Map at Boot up.

Address	Task	Size
9000:???? 9000:0000	for “System Initialization routine”	
0000:7DFF 0000:7C00	“Disk Boot Strap Routine”	512b
0000:04FF 0000:0400	BIOS Data Area	256b
0000:03FF 0000:0000	Interrupt Vector Table(real mode)	1Kb

8086 real mode addresses

4.2. SYSTEM INİTİALİZATİON ROUTİNE

This part of the system is loaded by the “Boot Strap Routine”. “Boot Strap Routine” leaves control to this routine. The tasks of this part heavily complex.

1. Investigates the hardware and bios data.
    - identifies CPU.
    - find port addresses for devices
    - get system information using bios functions
    - get memory size
2. Loads Kernel and API from boot disk to memory at proper locations.
    - load memory images for Kernel and API using bios interrupt 13h.
3. Initialize and Enter “Protected Mode”.
    - enable A20 gate.
    - initialize descriptor tables
    - switch CPU to “protected mode”
    - form page directory table and page tables
    - enable “paging”
    - start first task and set TSS.
4. Initialize the Kernel and API.
    - initialize runtime properties of the Kernel and API
5. Load and Run Command Interpreter.
    - load using API and give control to it.

Address	Usage	Size
	Free Physical Memory for Applications	
00101000	Page Tables	**
00100FFF 00100000	Page Directory Table	4Kb
000FFFFF 000C0000	ROM	
000B8FFF 000B8000	Color Text Screen Memory	
000B7FFF 000B0000	BW Text Screen Memory (not supported)	
000AFFFF 000A0000	Graphics Screen Memory	
0009FEFF 00090000	System Initialization Routine	64Kb*
0008FFFF 00040000	API Code and Data	320Kb
0003FFFF 00010000	Kernel Code and Data	192Kb
0000FFFF 00001000	Global Descriptor Table (max. 7680 entry)	60Kb
00000FFF 00000900	StartTech System Data Area	1792b
000008FF 00000800	copy of bios system data at 0400-04FF	256b
000007FF 00000000	Interrupt Descriptor Table(256 entry)	2Kb

(32 bit physical addresses)
(*) Maximum size
(**) Determined from the size of the virtual memory.

StartTech System Data Area (at 00000900h)

Offset (hex)	Description	Size
00	CPU type	WORD 7: 80386 8: 80486 9: Pentium or better
02	Coprocessor type	WORD
04	Physical Memory Size	DWORD
08	Virtual Memory Size	DWORD
0C	User Memory Start	DWORD
10	User Physical memory Size	DWORD
14	User Virtual memory Size	DWORD

4.3. SYSTEM KERNEL

4.3.1. PROCESSOR SCHEDULER

What is a Process in StarTech®?

In StarTech®, a process is whole the code, data, stack and any resources allocated for it. Every process has at least one thread called main thread. If necessary one or more threads may be created using proper API functions.

System Level Attributes of a Process:
1- Owner Process ID.(Index in Process_List)
2- Process Attributes word

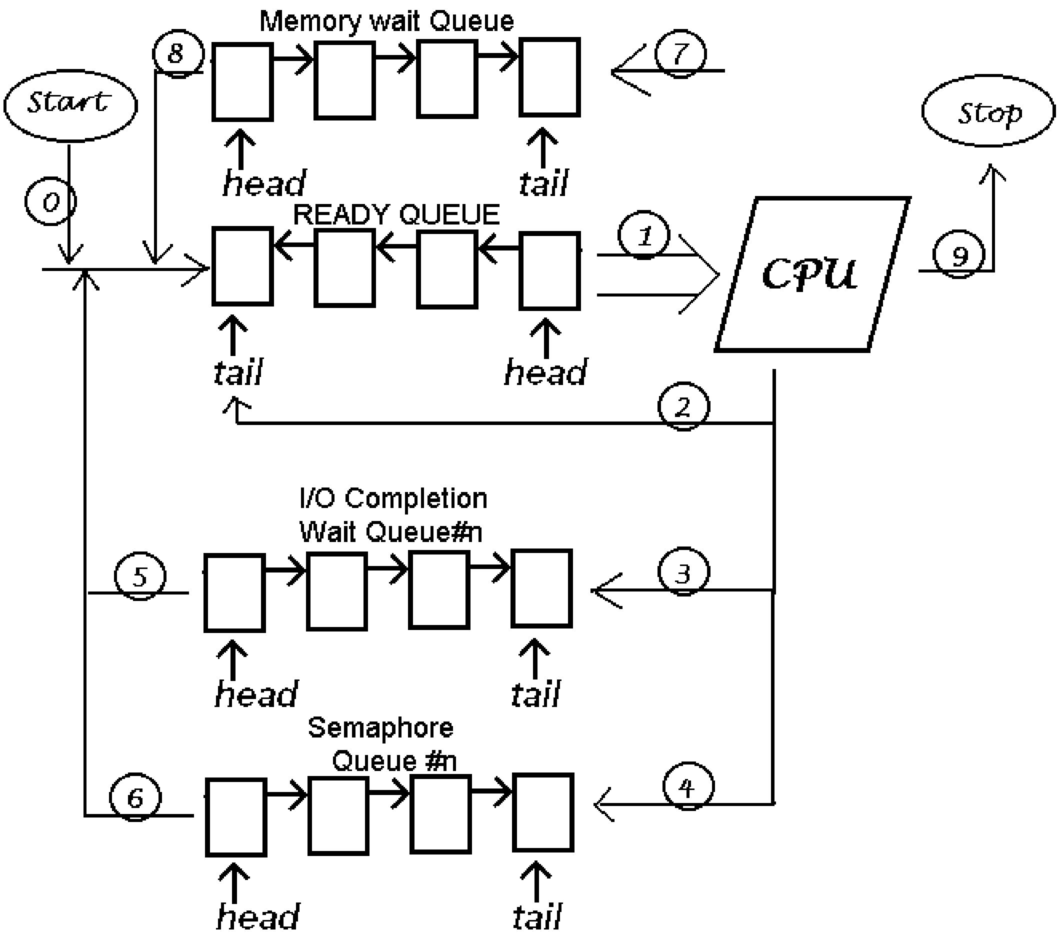
What is a Thread?

A thread is an execution path for a process. This means it uses the common code and data with other threads in the same process but has separate stack, and copy of CPU registers. Threads are atoms of processes.
Task Switches are also made on the threads not on the process level.

System Level Attributes of a Thread:
1- Process ID in which the thread is
2- Selector for TSS of the thread

TSS is Task State Segment contains everything about a thread. Most of it required by Intel 80386 architecture. Other parts are a link to previous TSS, status of the thread, thread ID, process ID, some thread attributes.

Processor scheduler of StarTech® uses round robin scheduling algorithm which is a time slice based preemptive method for multitasking.



Flow of the threads in Process Scheduler part of the StarTech System.

NO **FUNCTION NAME** **TASK**

() INIT_SCHEDULER Initialize the process scheduler

input	none
output	none

- Thread Management

(0*) CPU_CREATE_THREAD Allocates a new thread ID in thread_list

input	eax = process id
output	eax = thread id ebx = TSS selector

(0) CPU_ADD_THREAD Puts a new thread to ready queue

input	eax = thread id
output	none

CPU_SCHEDULER Performs a task switch

input	none
output	none

(2) CPU_MOVETO_READY_LIST Moves current thread to ready queue

input	none
output	none

(3) CPU_MOVETO_IO_LIST Moves current thread to a I/O queue

input	eax = device id
output	none

(4) CPU_MOVETO_SEMAPHORE Moves current thread to a semaphore queue

input	eax = semaphore id
output	none

(5) CPU_IO_DISPATCHER Dispatch an I/O head thread to ready queue

input	eax = device id
output	eax = status

(5*) CPU_WAIT_DISPATCHER Dispatch an I/O queue thread to ready queue

input	eax = device id ebx = thread id
output	eax = status

(6) CPU_SEMAPHORE_DISPATCHER Dispatch a semaphore head thread to ready queue

input	eax = semaphore id
output	eax = status

(7) CPU_SWAPOUT_THREAD Moves a ready queue thread to memory wait queue

input	eax = thread id
output	eax = status

(8) CPU_SWAPIN_THREAD Moves a memory wait queue thread to ready queue

input	eax = thread id
output	eax = status

(9) CPU_TERMINATE_THREAD Clears a thread from thread_list and its queue

input	eax = thread id
output	eax = status

() CPU_CHECK_THREAD_STATE

input	eax = thread id
output	eax = status

- Process Management

(0) CPU_CREATE_PROCESS Allocates a new process ID in proc_list

input	eax = owner's process id ebx = process attributes
output	eax =process id

(7) CPU_SWAPOUT_PROCESS Swaps out all the threads of a process

input	eax = process id
output	none

(8) CPU_SWAPIN_PROCESS Swaps in all the threads of a process

input	eax = process id
output	none

(9) CPU_TERMINATE_PROCESS Clears a process form proc_ist

input	eax = process id
output	none

() CPU_CHECK_PROCESS_STATE

input	eax = process id
output	none

4.3.2.SYNCHRONIZATION SYSTEM

StarTech® uses semaphores to implement synchronization system. There is an array of semaphores some of which are allocated for special purposes such as system devices. These semaphores are counting semaphores and these functions are uses processor scheduler functions to implement wait state.

SYNC_INIT initialize synchronization system

input	none
output	none

SYNC_CHECK test to see if a semaphore is available

input	eax = semaphore id
output	eax = status

SYNC_WAIT grant to access a semaphore if it is available

input	eax = semaphore id
output	none

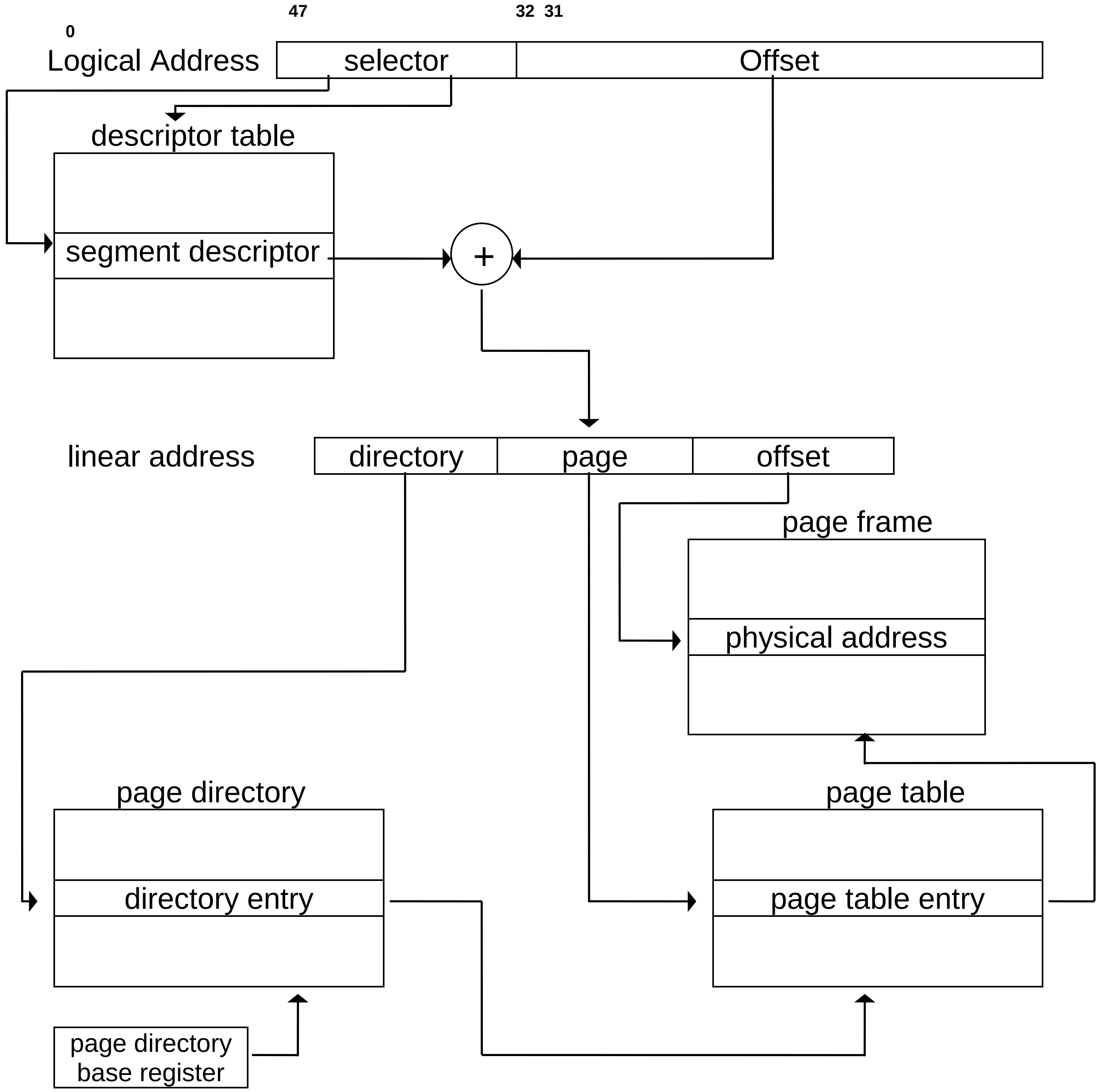
SYNC_SIGNAL signals freeing of a semaphore

input	eax = semaphore id
output	none

SYNC_SET sets a semaphore to a value

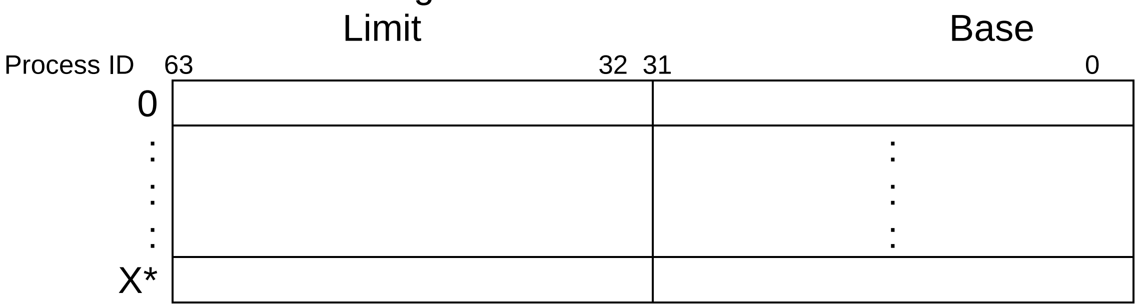
input	eax = semaphore id ebx = value
output	none

4.3.3. MEMORY MANAGEMENT



Intel 80386 address translation diagram

Page Frame Allocation Table



* This table will be dynamic so size of the table will be calculated according to physical memory size. Maximum size can be 128 Kb.

Physical memory and swap-file usage is performed by a bit-string. Each bit in these bit-strings indicates a 4Kb page used or not.

FUNCTION NAME **TASK**

MEM_VIRTUAL_ALLOC Allocates 4KB page frames

input	eax = process id ebx = number of 4K pages to allocate
output	eax = address of first page on error :eax=0

MEM_VIRTUAL_DEALLOC frees page frames.

Input	eax = process id
output	eax = status

MEM_MEM_TO_SWAPFILE swaps out a 4Kb from memory to disk

input	none
output	none

MEM_SWAPFILE_TO_MEM swaps in a 4Kb from disk to memory

input	input from control register
output	none

4.3.4.SYSTEM DEVICE DRIVERS

4.3.4.1.KEYBOARD & DISPLAY FUNCTIONS

FUNCTION NAME | **TASK**

IO_CON_INIT_PROC — clears a process' keyboard buffer and screen buffer

input	eax = process id
output	none

IO_CON_GET_CON_PROCESS — get current concole process

output	none
output	ax = con process

IO_CON_SET_CON_PROCESS — set current console process

input	ax = con process
output	none

IO_CON_KBD_READ — gets a character from keyboard buffer

input	eax = process id
output	ah = Scan Code al = ASCII code

IO_CON_KBD_STS — gets shift keys status

input	none
output	ax = shift status

IO_CON_KBD_CLR — used to clear kbd buffer

input	eax = process id
output	none

IO_CON_SCR_WRITE — puts a character on screen buffer

input	eax = process id bl = ASCII character bh = color
output	none

IO_CON_SCR_CLR — clears screen buffer

input	eax = process id
output	none

IO_CON_SCR_SET_CUR set cursor postion.

input	eax = process id bx = position
output	none

IO_CON_SCR_GET_CUR get cursor position

input	eax = process id
output	bx = position

IO_CON_SCR_SWAP copy process' screeen bufffer to screen buffer

input	eax = process id
output	none

IO_CON_SCR_SET_BUFFER set a process' screen buffer pointer

input	eax = process id bx=selector edx=offset
output	none

4.3.4.2.FLOPPY DISK FUNCTIONS

IO_FDD_INIT initialize flopyy controller

input	none
output	none

IO_FDD_READ read a sector from floppy disk

input	eax = thread id ebx = drive no ecx = Linear Sector Address(LSA) es:edi = buffer
output	ax =status

IO_FDD_WRITE writes a sector to a floppy disk

input	eax = thread id ebx = drive no ecx = Linear Sector Address(LSA) es:edi = buffer
output	ax =status

IO_FDD_STATUS gets status of alast operation performed

input	eax = drive no
output	ax = status

4.3.4.3. Hard Disk Functions

IO_HDD_INIT — Initialize hard disk controllers

input	none
output	none

IO_HDD_READ — read a sector into memory

input	eax = thread id ebx = drive no ecx = Linear Sector Address(LSA) es:edi = buffer
output	ax =status

IO_HDD_WRITE — write a sector to hard disk

input	eax = thread id ebx = drive no ecx = Linear Sector Address(LSA) es:edi = buffer
output	ax =status

IO_HDD_STATUS — get hard disk status

input	eax = drive no
output	ax = status

4.3.4.4. Parallel Port Functions

IO_PRN_INIT reset printer

input	al = printer no
output	ah =status

IO_PRN_WRITE sends a character to printer

input	al = printer no ah = character
output	ah =status

IO_PRN_STATUS returns the state of the printer

input	al = printer no
output	ah =status

4.3.4.5. COMM. PORT FUNCTIONS

IO_COMM_INIT reset the port and data buffer

input	al = comm port no
output	ah =status

IO_COMM_RECIEVE retrieves a char from the data buffer

input	al = comm port no
output	al = char ah = status

IO_COMM_SEND sends a char to comm port

input	al = comm port no ah = char
output	ah =status

IO_COMM_STATUS returns the state of the

input	al = comm port no
output	ah =status

4.4. APPLİCATİON PROGRAM INTERFACE

4.4.1. Process Execution System

CreateProcess()

Specification:	Starts execution of a process.	
INPUT		
CommandLine	pSTR	Program file name, path

PROCID CreateProcess(pSTR CommadLine);

Returns: process id (non zero) if successful, zero otherwise.

TerminateProcess()

Specification:	Ends execution of a process.	
Input:		
ProcessID	PROCID	Process ID of process to end
ReturnStatus	DWORD	Return code for return

```
BOOL TerminateProcess( PROCID      ProcessID,
                       DWORD       ReturnStatus);
```

Returns: TRUE if successful, FALSE otherwise.

ExitProcess()

Specification:	Ends execution of a process.	
Input:		
ReturnStatus	DWORD	Return code for return

DWORD ExitProcess(DWORD   ReturnStatus);

Returns to the operating system.

Wait()

Specification:	Suspends a task for some specified time period.	
Input:		
Period	DWORD	Delay in mili seconds

void Wait(DWORD Period);
Returns: nothing

♦

4.4.2. Synchronization System

CreateSemaphore()

Specification:	Creates a semaphore	
Input:		
SemaphoreName	pSTR	Name of the semaphore
InitCount	DWORD	Initial semophore count.

```
SEMAPHORE  CreateSemaphore(      pSTR  SemaphoreName,
                                 DWORD      InitCount);
```

Returns:
    on success: Semaphore ID,
    on failure   :   zero.

DeleteSemaphore()

Specification:	deletes a semaphore.	
Input:		
SemophoreID	SEMAPHORE	Semaphore to wait.

```
void DeleteSemaphore(    SEMAPHORE        SemaphoreID);
```

GetSemaphoreID()

Specification:	Gets a semaphore’s ID	
Input:		
SemaphoreName	pSTR	Name of the semaphore

```
SEMAPHORE  GetSemaphoreID(pSTRSemaphoreName);
```

Returns:
    on success: Semaphore ID,
    on failure   :   zero.

WaitSemaphore()

Specification:	Suspends a task until an event occurs or time out. If successful then decreases semaphore	
Input:		
SemophoreID	SEMAPHORE	Semaphore to wait.

BOOL WaitSemaphore(SEMAPHORE SemaphoreID);

Returns: TRUE if event occurs, FALSE otherwise.

ReleaseSemaphore()

Specification:	Releases a semaphore.(increments)	
Input:		
SemophoreID	SEMAPHORE	Semaphore to wait.

BOOL ReleaseSemaphore(SEMAPHORE SemaphoreID);

Returns: TRUE if successful, FALSE otherwise.

♦

4.4.3. Interprocess Communication System

CreateMailBox()

Specification:	Creates a mail box.	
Input:		
MailBoxName	pSTR	Mail box name.
Size	DWORD	Size of mailbox in bytes

```
MAILBOX CreateMailBox( pSTR   MailBoxName,
                       DWORD      Size);
```

Returns:
on success ID of mailbox,
otherwise zero.

GetMailBoxID()

Specification:	Gets a mail box ID.	
Input:		
MailBoxName	pSTR	Mail box name.

```
MAILBOX GetMailBoxID(  pSTR  MailBoxName);
```

Returns:
on success ID of mailbox,
otherwise zero.

GetMailBoxInfo()

Specification:	Gets information about a mail box.	
Input:		
MailBoxID	MAILBOX	Mail box ID.
MailInfo	DWORD *	Mail box info structure.

```
BOOL GetMailBoxInfo(   MAILBOX          MailBoxID,
                       DWORD            *MailInfo);
```

Returns: TRUE if successful, FALSE otherwise.

SetMailBoxInfo()

Specification:	Sets a mail box state.	
	Input:	
MailBoxID	MAILBOX	Mail box ID.
MailInfo	DWORD	Mail box info structure.

```
BOOL SetMailBoxInfo(    MAILBOX        MailBoxID,
                        DWORD          MailInfo);
```

Returns: TRUE if successful, FALSE otherwise.

SendMail()

Specification:	Send a mail.	
	Input:	
MailBoxID	MAILBOX	Mail box ID.
Data	pVOID	data to send
Size	DWORD	size of data
How	DWORD	Actions to take

```
BOOL SendMail(    MAILBOX    MailBoxID,
                  pVOID      Data,
                  DWORD      Size,
                  DWORD      How);
```

Returns: TRUE if successful, FALSE otherwise.

GetMail()

Specification:	Get a mail.	
	Input:	
MailBoxID	MAILBOX	Mail box ID.
Data	pVOID	data to recieve
Size	DWORD	size of data
How	DWORD	Actions to take

```
BOOL GetMail(     MAILBOX    MailBoxID,
                  pVOID      Data,
                  DWORD      Size,
                  DWORD      How);
```

Returns: TRUE if successful, FALSE otherwise.

4.4.4. FILE SYSTEM

StarTech® File System (STFS)

StarTech® File System (STFS) specifications:

File System Structure : indexed allocation using linked scheme
Directory Structure : tree-structured directories
Directory Implementation : using linear list
Free Space Management : using bit vector

Cluster Size= 4 KB = 8 Sector
Sector Size = 512 Byte

Disk Usage:

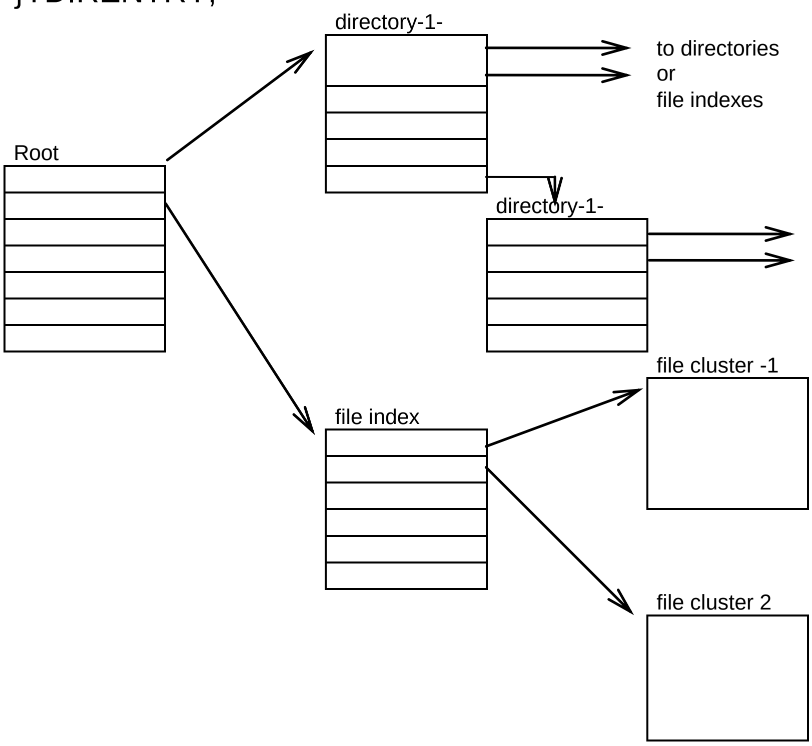
Sector #	**Usage**	
0	Boot Sector	
1 - 160	Bit Vector	
161 - 168	Root Entry Table	cluster #1
169 - 176	Free disk space	cluster #2
... - ...	"	"
... - ...	"	"
... - ...	"	"
... - ...	"	"
... - ...	"	cluster #n
...	Swap Disk Area	excluded from file system

Directory Entry Table:

Entry[0]	First Entry	64 byte
Entry[1]		"
Entry[2]		"
Entry[3]		"
...		"
		"
		"
Entry[62]	Last Entry	"
NextDirPtr	Pointer to next entry table for this directory(if any)	"

Directory Entry Structure :

```
typedef struct {
  WORD      Type;                    // 0x0000 - unused entry
                                     // 0x0001 - directory entry
                                     // 0x0002 - file entry
  WORD      Attrib;                  // file attributes - not used for directories -
                                     // 0x0001 - archive
                                     // 0x0002 - hidden
                                     // 0x0004 - read-only
                                     // 0x0008 - system
                                     // 0x0010 - executable
                                     // 0x0020 - binary file
                                     // 0x0040 - ASCII file
  char      Name[21];                // 20 char long name
  char      Ext[6];                  // 5 char long extension
  DWORD     FileSize;                // upto 4GB file
  DWORD     BlockCount;              // number of clusters allocated (not used for dirs)
  DWORD     Pointer;                 // pointer to next data item cluster no 1 - xxx
                                     // A 0 means end.
  FILE_DATETIME CreateDate;          // Cretion date
  FILE_DATETIME ModifyDate;          // Modify date
  FILE_DATETIME AccessDate;          // Acess date
}TDIRENTRY;
```



CreateFile()

Specification:	Creates a file	
	Input:	
FileName	pSTR	filename to created

BOOL CreateFile (pSTR filename);

OpenFile()

Specification:	grants access to a file	
	INPUT	
FileName	pSTR	file name
FileMode	DWORD	file parameters can be F_READ 0x00000000 F_WRITE 0x00000001

```
HFILE OpenFile(    pSTR        FileName,
                   DWORD       FileMode);
```

Returns:

On success, non-zero value, a file handle,Otherwise, zero

CloseFile()

Specification:	closes a file	
	INPUT	
FileHandle	HFILE	an open file handle

```
BOOL CloseFile(    HFILE       FileHandle);
```

Returns:

On success, TRUE
Otherwise, FALSE

ReadFile()

Specification:	reads from a file to a buffer	
INPUT		
FileHandle	HFILE	an open file handle
BufferSize	DWORD	# of bytes to read
Buffer	pVOID	input buffer for read

```
DWORD ReadFile( HFILE     FileHandle,
                DWORD     BufferSize,
                pVOID     Buffer);
```
Returns: # of bytes read

WriteFile()

Specification:	writes to a file	
INPUT		
FileHandle	HFILE	an open file handle
BufferSize	DWORD	# of bytes to write
Buffer	pVOID	output buffer for write

```
DWORD WriteFile( HFILE     FileHandle,
                 DWORD     BufferSize,
                 pVOID     Buffer);
```
Returns: # of bytes written

EndOfFile()

Specification:	see if end of file	
INPUT		
FileHandle	HFILE	an open file handle

BOOL EndOfFile(HFILE fh);

SeekFile()

Specification:	positions read/write head on a position in a file	
INPUT		
FileHandle	HFILE	an open file handle
Offset	DWORD	offset from From
From	BYTE	SEEK_SET 0 from beggining of the file SEEK_CUR 1 “ current position SEEK_END 2 “ end of file

BOOL SeekFile(HFILE FileHandle, DWORD Offset);

Returns:
On success, TRUE
Otherwise, FALSE

GetFileInfo()

Specification:	get attributes of a file	
INPUT		
FileName	pSTR	file name
OUTPUT		
FileAttrib	pFILEATRB	file attributes

BOOL GetFileInfo(pSTR FileName,
pFILEATRB FileAttrib);

Returns:
On success, TRUE
Otherwise, FALSE

SetFileAttrib()

Specification:	set attributes of a file	
INPUT		
FileName	pSTR	file name
FileAttrib	DWORD	file attributes

BOOL SetFileAttrib(pSTR FileName,
DWORD FileAttrib);

Returns:
On success, TRUE
Otherwise, FALSE

FileSearch()

Specification:	searches current directory for a file or directory.	
INPUT		
FileName	pSTR	file name

BOOL FileSearch(pSTR fname);

Returns:
On success, TRUE
Otherwise, FALSE

FileList()

Specification:	returns specified file in the	
INPUT		
Path	pSTR	path
EntryNo	DWORD	index
OUTPUT		
FileInfo	pFILEATRB	file information

```
BOOL FileList(   pSTR        Path,
                 DWORD       EntryNo,
                 pFILEATRB   FileInfo);
```

Returns:
On success, TRUE
Otherwise, FALSE

Remove()

Specification:	delete file or directory	
INPUT		
FileName	pSTR	file name

BOOL Remove(pSTR FileName);

Returns:
On success, TRUE
Otherwise, FALSE

Rename()

Specification:	rename a file or directory	
INPUT		
OldName	pSTR	file name
NewName	pSTR	file name

```
BOOL Rename(    pSTR OldName,
                pSTR NewName);
```

Returns:
    On success, TRUE
    Otherwise, FALSE

CreateDir()

Specification:	creates a directory	
INPUT		
DirName	pSTR	directory name

```
BOOL CreateDir(pSTR DirName);
```

Returns:
    On success, TRUE
    Otherwise, FALSE

GetCurDir()

Specification:	gets current directory	
OUTPUT		
DirName	pSTR	directory name

```
void GetCurDir(pSTR DirName);
```

SetCurDir()

Specification:	sets current directory	
	INPUT	
DirName	pSTR	directory name

BOOL SetCurDir(pSTR dirname);

Returns:
    On success, TRUE
    Otherwise, FALSE

DiskFree()

Specification:	gets disk space in bytes	
	INPUT	
DriveNo	DWORD	drive no

DWORD DiskFree(DWORD driveno);

Returns: Free disk space in bytes

DiskSize()

Specification:	gets disk space in bytes	
	INPUT	
DriveNo	DWORD	drive no

DWORD DiskSize(DWORD driveno);

Returns: all disk space in bytes

♦

4.4.5. COMMUNICATIONS

OpenComm()

Specification:	grants access to a comm port	
INPUT		
CommNo	WORD	comm port no
CommSettings	pCOMMSTRC	communication parameters structure

```
BOOL OpenComm(WORD         CommNo,
              pCOMMSTRC    CommSettings);
```

Returns:
    On success, TRUE
    Otherwise, FALSE

SendComm()

Specification:	send a string of chars to comm port	
INPUT		
CommNo	WORD	comm port no
Buffer	pVOID	buffer to send
BufLen	DWORD	length of the buffer

```
DWORD SendComm(   WORD     CommNo,
                  pVOID    Buffer,
                  DWORD    Buflen);
```

Returns:
    Number of bytes sent.

ReceiveComm()

Specification:	Gets a string of chars from a comm port	
INPUT		
CommNo	WORD	comm port no
BufLen	DWORD	length of the buffer
OUTPUT		
Buffer	pVOID	buffer

```
DWORD ReceiveComm(  WORD     CommNo,
                    pVOID    Buffer,
                    DWORD    Buflen);
```

Returns: Number of bytes received.

CloseComm()

Specification:	releases access to a comm port	
INPUT		
CommNo	WORD	comm port no

BOOL CloseComm(WORD CommNo);

Returns:
On success, TRUE
Otherwise, FALSE

OpenPrinter()

Specification:	grants access to a printer	
INPUT		
PrnNo	WORD	printer no

BOOL OpenPrinter(WORD PrnNo);

Returns: On success, TRUE, otherwise, FALSE

SendPrinter()

Specification:	send a string of chars to printer	
INPUT		
PrnNo	WORD	printer no
Buffer	pVOID	buffer to send
BufLen	DWORD	length of the buffer

```
DWORD SendPrinter(   WORD    PrnNo,
                     pVOID   Buffer,
                     DWORD   Buflen);
```

Returns: Number of bytes sent.

ClosePrinter()

Specification:	closes a printer	
INPUT		
PrnNo	WORD	printer no

BOOL ClosePrinter(WORD PrnNo);

4.4.6. DISPLAY/KEYBOARD I/O

GetCh()

Specification:	gets a character from keyboard (no echo)

WORD GetCh(VOID);

Returns:
    a character from keyboard

GetChe()

Specification:	gets a character from keyboard and echos

char GetChe(VOID);

Returns : a character from keyboard

GetStr()

Specification:	gets a string from keyboard	
	INPUT	
InStr	pSTR	string to be read.

void GetStr(pSTR InStr);

PutCh()

Specification:	puts a character to display.	
	INPUT	
OutCh	char	char to be written.

void PutChar(char OutCh);

PutChClr()

Specification:	puts a character to display.	
	INPUT	
OutCh	char	char to be written.
Color	WORD	color attributes

void PutCharClr(char OutCh, WORD Color);

PutStr()

Specification:	puts a string to display	
INPUT		
OutStr	pSTR	string to be written.

void PutStr(pSTR OutStr);

PutStrClr()

Specification:	puts a string to display	
INPUT		
OutStr	pSTR	string to be written.
Color	WORD	color attributes

void PutStrClr(pSTR OutStr, WORD Color);

GetShiftKeys()

Specification:	gets status of shift keys

WORD GetShiftKeys(void);

Returns: shift keys status

ClrScr()

Specification:	Clears Screen

void ClrScr(void);

GotoXY()

Specification:	position the cursur	
INPUT		
Column	BYTE	X -coordinate(0-79)
Row	BYTE	Y- coordinate(0-24)

void GotoXY(BYTE Column,BYTE Row);

WhereX()

Specification:	gets current cursor X position

BYTE WhereX(void);

Returns:
cursors X position.

WhereY()

Specification:	gets current cursor Y position

BYTE WhereY(void);

Returns:
cursors Y position.

GetConsoleProcess()

Specification:	gets current console process' process id

DWORD GetConsoleProcess(void);

Returns:
current console process' process id.

SetConsoleProcess()

Specification:	Copies process screen buffer to physical screen and sets it current console process	
INPUT		
ProcID	DWORD	process id

void SetConsoleProcess(DWORD ProcID);

♦

4.4.7. Misc. Functions

SystemVersion()

Specification:	returns system version e.g. 100 means 1.00

DWORD SystemVersion(void)

CPUType()

Specification:	gets CPU and coprocessor type

DWORD CPUType(void)

PhysicalMemory()

Specification:	gets physical memory size

DWORD PhysicalMemory(void)

VirtualMemory()

Specification:	gets virtual memory size

DWORD VirtualMemory(void)

GetDateTime()

Specification:	gets system date and time	
OUTPUT		
DateTime	pTDATETIME	current time data

void GetDateTime(pTDATETIME datetime)

SetDateTime()

Specification:	Sets system date and time	
INPUT		
DateTime	pTDATETIME	time data

void SetDateTime(pTDATETIME datetime)

4.5.SYSTEM COMMAND INTERPRETER

Command interpreter is the user interface of StarTech®. It is a text mode simple user interface that enables users to run programs, terminate them and perform disk operations on file system.

User commands:

(English)	(Turkish)	Explanation
attrb	oz	Sets attribute of a file
cd	kd	Changes current directory
clr	t	Clears screen
c	k	Copies a file
date	t	Displays and changes date
dd	sk	Deletes a directory
dl	sd	Deletes a file
l	l	Lists current directory
h or ?	y or ?	Displays help
nd	yk	Creates a new directory
m	hf	Displays memory sizes
p	yl	Displays current directory
prn	yaz	Prints a file to printer
ren	id	Renames a file
rend	idk	Renames a directory
time	z	Displays and changes time
ver	sur	Displays Current System version

4.6.SYSTEM TOOLS & APPLİCATİONS.

Application programs will be developed using C and StarTech® API library functions. Following steps should be performed.

1. API function library header file must be included.
2. Source must be compiled using an 32 bit DOS compiler.
   If BCC32 used source must be compiled to assembly first then it requires these changes.
   a. remove .FLAT directive
   b. add ASSUME CS:_TEXT,DS:_DATA
   c. compile it using TASM

   If WCC386 is used, it must be compiled using small model for switch(-ms).
3. Link object code with library object code and start up object files.
4. Convert produced .EXE file to StarTech executable file format using “MakeStar.exe” utility.

5. Source Codes

Source listings are on separate and continous pages.

BootStrap:
    Source\BootStrp\Bootn.asm
System Initialization:
    Source\SysInit\Sysinit.asm
Kernel:
    Source\Kernel\Init\Kernel.asm
    Source\Kernel\Init\˜Kernel.asm
    Source\Kernel\Process\Cpusched.asm
    Source\Kernel\Synchon\Syncsys.asm
    Source\Kernel\Memory\Memory.asm
    Source\Kernel\Device\Device.asm
    Source\Kernel\Device\Console.asm
    Source\Kernel\Device\Floppy.asm
    Source\Kernel\Device\Hdd.asm
    Source\Kernel\Device\Printer.asm
    Source\Kernel\Device\Comm.asm
API:
    Source\API\ALL\API.c
    Source\API\ALL\Api_end.asm
    Source\API\Process\Proc_api.c
    Source\API\SyncSys\ Syncsys.c
    Source\API\IPC\ipc.c
    Source\API\FileSys\Filesys.c
    Source\API\CommPrn\Comprn.c
    Source\API\Console\Console.c
    Source\API\Misc\Misc.c

Command Interpreter:
    Source\Command\Command.c

Tools & Applications:
    Source\App\Edit.c

Development Tools:

reloc2.c	reloc2.exe	EXE file relocator (Updated to handle files more than 64K)
setloc.c	setloc.exe	sets an EXE's relocation items to a value
putfile.cpp	putfile.exe	Copies files to Disk Sectors (Updated to handle files more than 64K)
sectorc.c	sectorc.exe	Reads contents sectors from disk
makestar.c	makestar.exe	(Post-compiler) converts a DOS 32bit EXE to StarTech® executable format

CONCLUSİON

The project has just finished. We have finished coding and paritally tested it. Kernel is programmed using 80386 assembly and partially 80386 machine language and API is coded with C and inline assembly. Alpha test has been performed on finished parts of the system.

The current code is useful both as a tool for operating system development and for exploration of the Intel architecture. We discuss in this section several enhancements that are necessary or interesting extensions of the current work.

1. Direct Calls to Kernel:
In the present design, all access to kernel via API for applications. ın some cases like console device drivers some functions can be called directly from kernel which would reduce call gate overhead two times.

2. Multi-thread support in API:
Currently processor scheduler part of the kernel supports thread based multi-tasking but API do not allow to create more than one thread per process. This may be changed by allocating LDT dynamically.

3. Cache for File System:
In STFS, there is no caching mechanism but this would be done at device driver level more efficiently. The cache would surely improve system performance greatly.

4. Local Memory Allocation:
There is no local memory management, it is supposed that proposed applications should request its data in their data segments. But this could be done dynamically.

REFERENCES

1. **IBM PC/AT Technical Reference**,IBM,1984

2. **INTEL 80386 PROGRAMMER'S REFERENCE MANUAL** ,Intel,1986

3. DONOVAN John J. - MADNICK Stuart E. :**Operating Systems** ,McGRAW-HILL Book Comp ,1986

4. DONOVAN John J. :**Systems Programming** ,McGRAW-HILL Book Comp ,1987

5. TANENBAUM Andrew S. :**Operating Systems:Design and Implementation**, Prentice-Hall,1987

6. ADAMS Phllip M. :**Writing DOS Device Drivers in C**, Prentice Hall,1990

7. NELSON Ross P. :**80386/80486 Programming Guide**, Microsoft Press,1991

8. ABEL Peter :**IBM PC Assembly Language and Programming**,Prentice-Hall,1991

9. DOUGLAS V.Hall :**Microprocessors and Interfacing**, **programming and hardware**, McGRAW-HILL,1992

10. GALVIN Silberschatz :**Operating System Concepts** ,Addison-Wesley,1994

11. SAATÇİ Ali :**Bilgisayar İşletim Sistemleri** ,Hacettepe Univ.

12. SCHILDT Hebert :**Windows 95 Programming in C and C++**,McGRAW-HILL,1995

13. **PC Intern**, McGraw-Hill,1995

14. SCHILDT Hebert :**C,The Complete Refercence** , McGRAW-HILL,1995

15. RUA Pınar, Öztürk Özgür :**PC'nin Sırları** , Sistem Yayıncılık,1995

APPENDİX

Real and Protected Modes.
Beginning from 80286, Intel CPUs have ability to work in Protected Mode (older CPUs have Real Mode only). For compatibility reasons, all CPUs start in Real Mode after reset. Below are presented main differences between Real Mode and Protected modes for Intel CPUs. Note there are: Real Mode, Protected Mode, Virtual 8086 Mode (they will be frequently called RM, PM, VM86, respectively; also 286+(386+) will mean Intel 80286(80386) or better).

There are some differences between these modes in memory addressing (PM can address all memory, while RM can't unless it is set in PM on 386+, and VM86 cannot unless using PM supporting it to remap memory - this way EMM386 works); instruction set (some instruction are not allowed in RM), privileges (something can be forbidden in PM for less privileged code, many operations are forbidden in VM86), interrupt handling. PM supports multitasking, PM can run tasks in VM86 (the VM86 cannot function alone, must have PM code supporting it; it works similarly 8086 CPU with few enhancements except interrupt servicing which goes through PM). PM cannot store data to code segment (unless by aliasing; MOV CS:[BX],AX is illegal in PM). VM86 and PM on 386+ can have selective I/O port access restrictions (some ports can be accessed without causing exception and other can't).
Memory addressing and Paging.

In any mode, opcode defines some offset and segment of referenced memory
address, e.g. mov ax,es:[bx+si+1] - segment es, offset bx+si+1, push si - segment ss, offset sp-2, opcode itself is referenced by segment cs and offset ip; the address is translated to Linear Address by adding the offset to base of the segment and the Linear Address is then translated to Physical Address which is outputed by CPU on its address pins.
In RM or VM86, the base is segment*10h; in PM the base is taken from
descriptor table (LDT or GDT) and can have any value.
The value in segment register is called "selector" and its bits 15-3 specify offset in LDT or GDT (the offset is multiply of 8), bit 2 is 0 for GDT, 1 for LDT, bits 1-0 specify RPL (Requested Privilege Level).
Unless Paging (possible in PM and VM86, on 386+ only) is enabled, Physical Address = Linear. With Paging, low 12 bits of Linear Address go to Physical, other are used as index to two-level page tables (first bits 31-22 select page directory, then bits 21-12 select page).
Paging can also restrict access to some pages (in a way non-privileged code can read it only or has no access at all), or define non-present pages which have assigned physical addresses and put in memory in a way

transparent to program when access to their Linear Address is attempted.

Note Linear Address space is 4GB on 386+, and probably no system has so much physical memory: Paging makes system able to simulate it has.

Segment has also limit. Initially, the limit is 0FFFFh for all segment registers and cannot be changed in RM or VM86. In PM it is loaded from LDT or GDT when segment register is loaded. On 286 in PM the limit can be up to 0FFFFh, on 386+ in PM it can be up to 0FFFFFFFFh.
Also, PM allows "expand down" segments which allow access from address
limit+1 to maximum possible value of limit (depend on segment type).

Privilege Levels and Rules.

In RM, CPU has full privileges. In PM and VM86, they can be restricted. This reduces possibility of making disasters by bad code.

Base rules: cannot access more privileged data or call less privileged code than own privilege (although can return to less privileged code).
Additional: call to more privileged code cannot use any target address caller wants, it can use addresses specified by system only; call to more privileged code must change stack to make sure enough stack space
is available for called code (so caller cannot cause crash in it).

There are 4 levels: level 0 is full privilege (except Debug Registers, which can be protected from access even from level 0; some instructions are reserved for level 0 only), the bigger level the less privileges are. Few terms used for Privilege Levels: CPL - Current PL, DPL - Descriptor PL, RPL - Requested PL (in selector), IOPL (in flags) -
max CPL allowing I/O sensitive opcodes (CLI, STI, PUSHF, POPF,...).

Unless accessing Conforming Code segment, privilege rules require max(CPL,RPL)<=DPL. To execute code (by FAR CALL or JMP) need DPL<=CPL
(note unless it is Conforming, must be DPL=CPL and RPL<=CPL) - cannot call less privileged procedure, for example. To transfer control to code with less PL (more privileged), must CALL via call gate (in such a case, need max(CPL,RPL)<=gate_DPL, but for code the gate refers to
may be code_DPL<gate_DPL; the gate is entry in GDT or LDT; privilege rules require also target_code_DPL <= CPL for CALL, = for JMP), this also requires TR to point to valid TSS because it switches stack: old SS:[E]SP are pushed on new stack, then parameters (as defined in call gate) are pushed, finally CS:[E]IP are pushed. On return from the call CPU detects RPL of CS on stack > CPL and switches stack back (if =, no stack switch, < inhibited by privilege rules), for proper functioning parameter counts on RET and in call gate must match. For stack segment DPL must be equal CPL (so in more privileged mode no crash is possible due to incorrect stack setting in less privileged, and in the less privileged there is no access to more privileged mode stack).

The RPL is for system to block possibility to pass a pointer from user

code which is invalid in user mode and valid in system: system uses RPL as for user code and gets access violation error in such a case.
It can be done using ARPL opcode which adjusts RPL for a selector, and sets ZF if changed (to inform OS invalid access might be attempted). OS uses it to set RPL of the pointer to CPL of the application code.

It is possible to check what access having to a segment by opcodes like VERR, VERW, LAR, LSL. They all set ZF if having access, clear if not.
First two simply verify R/W access, LAR gets bits defining access right for a segment, LSL gives the segment limit value. These opcodes allow checking what would cause access violation, instead getting the error.

Conforming code segments can be accessed without high privilege, they are for libraries which are shared between levels (otherwise would need keep separate copy for every level). Data kept in them can be accessed from any PL (providing they are readable) and code can be accessed (by
jump or call) from same or less privileged PL - in such a case CPL is NOT changed by the jump or call.
Cannot execute conforming code from more privileged PL: it is not trusteed enough to get CPL<DPL (greater privilege than defined in system tables).
I'm not sure how return from non-conforming to conforming code works, seems RPL taken from CS on stack determines new CPL (which may be less
privileged than the conforming code segment DPL).

Some instructions are allowed at CPL=0 only. They are:
Clear TaskÄSwitched Flag (CLTS), Halt Processor (HLT), loading some system registers (GDTR,IDTR,LDTR,MSW,TR), any access to CRx,DRx,TRx.
Some other require CPL<=IOPL. They are: IN, INS, OUT, OUTS, CLI, STI.
Also, POPF behavior depends on CPL: if CPL>0, IOPL and VM aren't changed by POPF, if CPL>IOPL, IF (interrupt enable) isn't changed.

Interrupts.

In every mode, there is an array containing information what action is to be taken in case of interrupt. Its first entry corresponds to INT 0, next to INT 1, and so on. It is called IDT(Interrupt Descriptor Table).
In RM, each entry in the IDT is simply far address of interrupt service routine. Initially IDT is located at address 0 and has 100h entries (400h bytes; some CPU-s have its limit 0FFFFh but the remainder isn't accessible in RM); on pre-80286 CPUs the IDT address and size cannot be changed, on 286+ can load and store them using LIDT and SIDT opcodes.

In PM the IDT has 8-byte entries which can be interrupt, trap or task gates. Trap differs from interrupt by leaving interrupt flag same as in interrupted code. Task gate causes calling another task. They all have DPLs and interrupt instruction causes General Protection error if CPL > interrupt or trap gate DPL. However, other interrupt sources have "CPL 0" - they can access any gate needed.

Some conditions can cause an Exception. They are (for 80386): divide error (0), debug exceptions (1), non-maskable interrupt (2), breakpoint

(3), overflow (4, on into opcode), bounds check (5, on bound opcode), invalid opcode (6), coprocessor not available (7), double fault (8,E), coprocessor segment overrun (9,P), invalid TSS (10,PE), segment not present (11,PE), stack error (12,E), general protection error (13,E), page fault (14,PE), coprocessor error (16); marked by P can occur in PM and VM86 only, marked by E push error code on stack if they occur in PM or VM86 (so stack is: error, IP, CS, flags; the error code is usually either 0 or selector causing the exception (in case selector is invalid or non-accessible), with flags on low order bits: bit 0 means external source, bit 1 IDT selector, bit 2 LDT; for page fault it is set of flags (bits 3-31 undefined): bit 0 set if page protection violation, 1 if writing, 2 if user mode), most of them push IP of opcode causing them, except 3,4,9 which push IP of next opcode.
Note: interrupt cannot be serviced at PL>CPL (unless via task switch), attempt to do it causes General Protection error.

Interrupt processing in PM is more complicated when interrupt handler has Privilege Level other than current code. It is handled similarly CALL via gate: stack is switched, new SS:SP are taken from TSS, old SS:SP are pushed on the new stack, then flags, CS, IP and eventually error code (for some exceptions) are pushed.
In VM86 interrupt pushes GS,FS,DS,ES,SS,ESP,EFLAGS,CS,EIP (exception also error code) onto PL 0 stack. There is VM bit in EFLAGS set to tell interrupt occured in VM86. Note IDT must contain task gates and 80386 trap or interrupt gates pointing to a non-conforming code segment with DPL=0 only - interrupt service must come through PL 0 or task switch. The VM86 itself has CPL 3 and is allowed in 386 task only.

Descriptor Tables (PM only).

Global Descriptor Table(GDT) can contain descriptors of any type except interrupt and trap gates. It is necessary for PM. First entry in GDT isn't used - it corresponds to null selector which can be loaded into segment register but causes exception if used for memory addressing.

Local Descriptor Table(LDT) can contain "normal" segment descriptors (not e.g. TSS) and call or task gates only. Usually every task has its own LDT (changed on task switch). The LDT must have descriptor in GDT.

Interrupt Descriptor Table(IDT) was discussed in "Interrupts" section.

"Normal" segment descriptors are referenced when a segment register is loaded and they describe a memory area and give some access to it.
Bit 2 of selector used selects table: 0 means GDT, 1 means LDT.
Other descriptors can be Task State Segment(TSS), and gates. They can be referenced "as a code segment", e.g. by far jump or call and they cause transferring control to task or code segment referenced by them.
It is kind of indirect jump or call (they contain target selector).
TSS or gate pointing to TSS cause task switch. Gate can be used to transfer control to more privileged code not accessible directly.
TSS can be also referenced by LTR (Load Task Register) opcode and it

is done once during PM initialization. LDT descriptor can be loaded into LDTR(register) by LLDT opcode and usually it is done once.

Segment and System Descriptors.

The following segment types (in byte [descriptor+5]) are supported (for all bit 7 means present in memory, bits 5-6 keep DPL which says what is maximum CPL which can access the descriptor, the restriction is for all descriptors, not segments only, except conforming segments):

10h+flags - data: bit 1 - writable, bit 2 - expand down
18h+flags - code: bit 1 - readable, bit 2 - conforming

for both, bit 0 is set by any access. The descriptor also contains limit in word [0] (in 386 segments extended to bits 0-3 of byte [6]) and base in bytes [2..4] (in 386 segments extended to byte [7]).
Byte [6] keeps few additional flags: bit 7 - granularity (limit is in 4kB pages; e.g. limit 0 means 0..0FFFh accessible), bit 6 - 32-bit addressing (applies to code and stack - use EIP, ESP, makes expand down segment upper limit 4GB), bit 5 must be 0, bit 4 is for programmer.

01h+flags - TSS: bit 1 - busy, bit 3 - 386 TSS
02h - LDT
04h+flags - call gate
05h - task gate
06h+flags - interrupt gate: bit 0 - trap, bit 3 - 386.

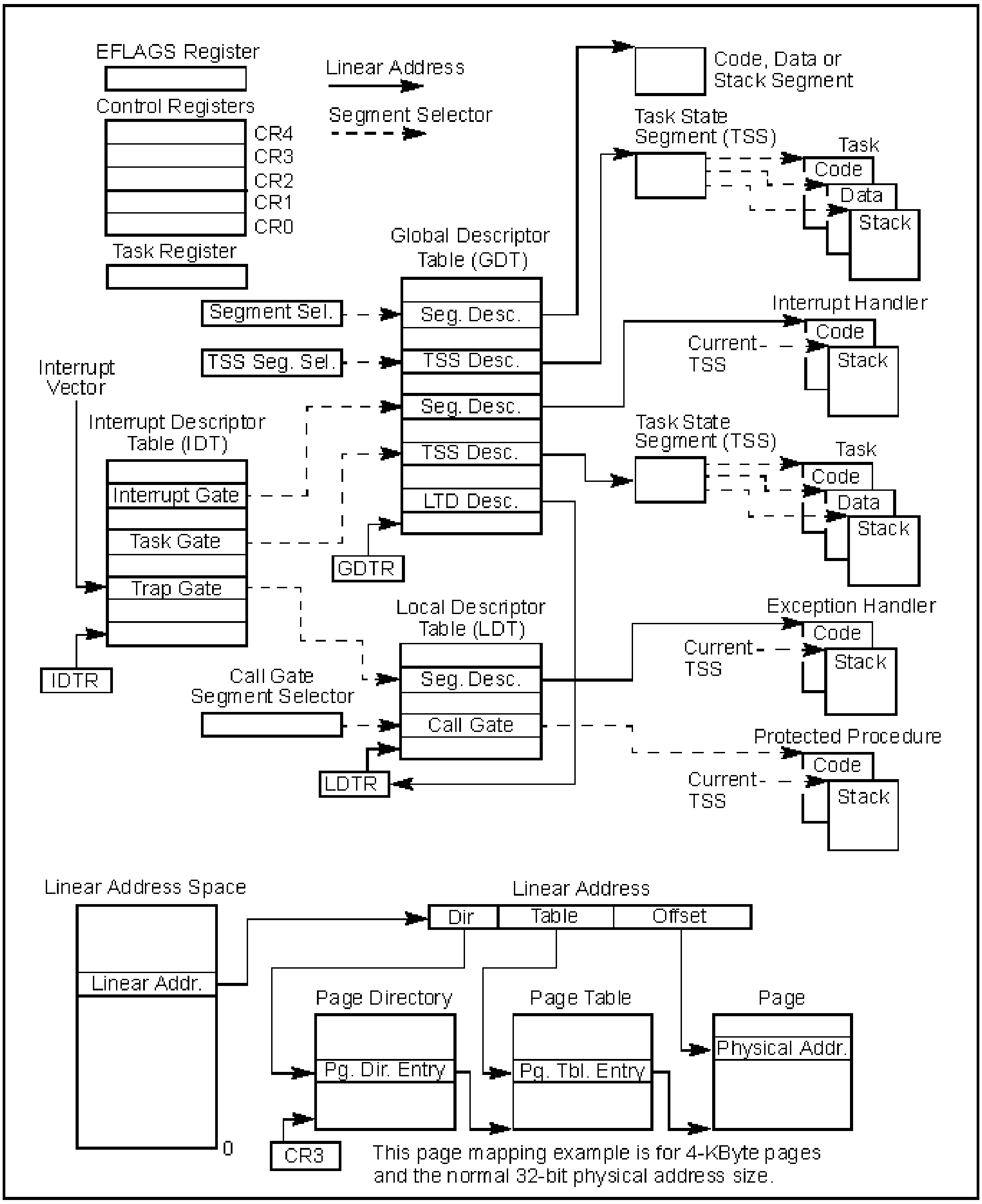
for all gates, word[2] keeps selector, word[0] and word[3] keep offset of called code (ignored for task gate), byte[4] keeps word count (0-31) for copying in case of inter-level call (call gate only, else ignored);
TSS and LDT have base and limit in same form as code and data segments have, they can have bit 7 set in byte [6] to specify limit in pages.
Word [6] should be 0 for the descriptor to mean the same on 286/386.

LDT is similar GDT, except not all descriptor types are allowed.
TSS holds entire task state (all registers: general, segment, flags, ip, ldtr); it also keeps link to caller TSS (valid if the task was activated by INT or CALL) and stacks (SS and [E]SP) for PL 0,1,2 (they are used when more privileged code is invoked via gate from less privileged). 386 TSS has also debug trap bit (if set, causes INT 1 on task switch to the TSS), I/O bit map (saying which I/O addresses can be accessed when CPL>IOPL without General Protection exception), and CR3 value for the task (can remap memory on task switch).

Page tables:

both page directory and page table entries keep referenced address in bits 31-12, have bits 11-9 reserved for programmer, must have bits 8,7, 4,3 set to 0; bit 5 is called A (accessed), it is set by CPU on access to the entry, bit 6 is called D (dirty), it is set if referenced memory is written; bit 0 is called P (present), all other are ignored if it is not set; bit 2 allows user (CPL=3) access if set, bit 1 allows user to write (together with bit 2 only), for CPL<3 read/write is allowed for any setting of bits 1 and 2 (no protection against system this way).

Note page table entries used are usually cached by CPU: modifying them
in memory may cause no mapping change until the cache is reloaded. The
cache is flushed every time CR3 (which points to first page directory entry) is loaded. Bits 0-11 of CR3 must be 0 (directory page-aligned).
Addressing through page tables:
CR3+(Linear_Address SHR 20) AND 0FFCh
is address in Page Directory, the entry at the address contains Page
Table address; Page Table address + (Linear_Address SHR 10) AND 0FFCh is address in Page Table and the entry at the address contains base
address of the page, combine it with bits 11-0 of Linear_Address and
the result is Physical Address. In case of any error, CR2 is set to the
Linear Address causing the error and error code explains what error.
Note: if Paging is enabled, CR3 must keep Physical Address of Page Directory and all other addresses are Linear Addresses.


EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Linear Address
Segment Selector
Code, Data or Stack Segment
Task State Segment (TSS)
Task
Code
Data
Stack
Global Descriptor Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LTD Desc.
GDTR
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
IDTR
Interrupt Handler
Current-TSS
Code
Stack
Task State Segment (TSS)
Task
Code
Data
Stack
Local Descriptor Table (LDT)
Call Gate Segment Selector
Seg. Desc.
Call Gate
LDTR
Exception Handler
Current-TSS
Code
Stack
Protected Procedure
Current-TSS
Code
Stack
Linear Address Space
Linear Addr.
0
Linear Address
Dir
Table
Offset
Page Directory
Pg. Dir. Entry
Page Table
Pg. Tbl. Entry
Page
Physical Addr.
CR3
This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.