Operating System
“Alt Ctrl Delete”
Technical product specification

x86

Version 0.01

Karol Gasiński
kuktus@gmail.com
www.mrkaktus.org

May 31, 2007

List of contents

1 System architecture

1.1 Introduction

Operating system Alt Ctrl Delete (in short ACD) is an experimental project which is aimed to be engaged in research at low-level computer device resource management. Because more functionality is transferred to user-level hosts, system is classified as µkernel (microkernel). All the code is compiled with NASM.

1.1.1 General assumption data

Performance	System is designed to achieve the maximum performance, so the kernel is entirely written in Assembler, and each function is repeatedly rewritten so as to achieve maximum optimization level.
Flexibility	All the system architecture is based on microkernel so as to achieve system containing great number of cooperating configurable modules. Owing to this structure, system may be optionally configured, so as to achieve maximum performance in any use.
Safety	In order to achieve proper protection for processes system access device mechanisms in x86/IA32 architecture. These are protected mode and memory paging.

1.1.2 Minimum system requirements

Owing to its compact and optimized structure, system can be started on nearly every low-profile computer:

- 80386-SX 16MHz processor or faster (coprocessor is not required).
- 1MB RAM.
- Video card compatible with VGA.
- 5.25” or 3.5” Floppy or any other removable.
- Keyboard.
- Monitor.

1.1.3 Limitations

At present system has a few limitations. However it will be all removed in the future:

- Support for up to 4 GB of system memory.
- Process code may take not more than 4MB.
- Memory available for the process up to (4GB – 4MB) including stack.
- Multithreading and multiprocessing is not supported.

1.2 System loading

Because of microkernel kernel type, ACD system must be loaded in specific way. Code which is loading kernel, in addition, loads boot data to the memory. These are previously defined during bootdisk configuration. These applications are owned by root, and they are treated like system's children.
This way of system boot makes it possible to configure system in full flexible way. E.g. you can load both kernel and process, whose task is to definite specific calculations. To apply this full calculating power is required. However keyboard server or network is not required.

1.2.1 Stage 1 – Bootloader

Loaded and BIOS-started bootloader, because of its very small size, is to load and start second stage bootloader (hereinafter SSBL – Second Stage Bootloader). SSBL has no limits considering its size, and may include all necessary procedures.

In order to load second stage, bootloader must use file system located on removable which was previously loaded from. For this purpose bootloader may load to memory definite FS management structures, which are then left for second-stage use. All the temporary data so as additional structures, or even the second stage, ale loaded below 639 Kilobytes of memory beginning from the highest level address.

Having configured its environment bootloader loads SSBL module and takes control to it. In case SSBL file is not found, the error message displays and the computer is suspended until the next system boot.

Adres początkowy	Rozmiar	Opis
00000 – 0KB – 000000	1KB	Tablica wskazów przerwań RM
00400 – 1KB – 001024	172B	BDA (Bios Data Area)
004AC – 1196B – 001196	68B	Zarezerwowane przez IBM
004F0 – 1264B – 001264	16B	Obszar komunikacji użytkownika
00500 – 1280B – 001280	2816B	Obszar danych (A)
01000 – 4KB – 004096	12KB	Obszar zmiany kontekstu. (B)
04000 – 16KB – 016384	15KB	——
07C00 – 31KB – 031744	0.5KB	Bootloader 1.
07E00 – 31.5KB – 032256	588.5KB	——
9B000 – 620KB – 634880	3.5KB	Bootloader 2. (C)
9BE00 – 623.5KB – 638464	4.5KB	Bootloader – FAT
9D000 – 628KB – 643072	4KB	Bootloader – Bufor odczytu
9E000 – 632KB – 647168	7KB	Bootloader – Root Dir
9FC00 – 639KB – 654366	1KB	EBDA (Extended BIOS Data Area)
A0000 – 640KB – 655360	64KB	Pierwszy ekran k.graficznej
B0000 – 704KB – 720896	32KB	Drugi ekran k.graficznej (jego 1 połowa) LUB 8 planów ekranu tekstowego (MONO)
B8000 – 736KB – 753664	32KB	Drugi ekran k.graficznej (jego 2 polowa) LUB 8 planów ekranu tekstowego (COLOR)
C0000 – 768KB – 786432	32KB	BIOS Karty Graficznej
C8000 – 800KB – 819200	160KB	BIOS Shadow Area
F0000 – 960KB – 983040	64KB	BIOS

Tab. State of first megabyte of RAM memory after Bootloader stops working.

1.2.2 Stage 2 – Second stage Bootloader

Second stage bootloader task is to load to memory and start main program which loads system (hereinafter SL - *System Loader*), and then mediate between himself and medium that system boots from. The main accessable function is ability to read any file to any memory address. To make it possible, second stage switches processor in Unreal Mode (Real Mode with addressing of full memory bus). Owing to apply this classification SL is universal multi-purpose for all data medium used for booting the system. Second stage bootloader stays in memory until SL stops working, applying its medium service.

tasks.

Adres początkowy	Rozmiar	Opis
00000 - 0KB - 000000	1KB	Tablica wskazów przerwań RM
00400 - 1KB - 001024	172B	BDA (Bios Data Area)
004AC - 1196B - 001196	68B	Zarezerwowane przez IBM
004F0 - 1264B - 001264	16B	Obszar komunikacji użytkownika (?)
00500 - 1280B - 001280	2816B	Obszar danych (A)
01000 - 4KB - 004096	12KB	Obszar zmiany kontekstu. (B)
04000 - 16KB - 016384	15KB	——
07C00 - 31KB - 031744	0.5KB	Bootloader 1.
07E00 - 31.5KB - 032256	584.5KB	——
9A000 - 616KB - 630784	4KB	System Loader
9B000 - 620KB - 634880	3.5KB	Bootloader 2. (C)
9BE00 - 623.5KB - 638464	4.5KB	Bootloader - FAT
9D000 - 628KB - 643072	4KB	Bootloader - Bufor odczytu
9E000 - 632KB - 647168	7KB	Bootloader - Root Dir
9FC00 - 639KB - 654366	1KB	EBDA (Extended BIOS Data Area)
A0000 - 640KB - 655360	64KB	Pierwszy ekran k.graficznej
B0000 - 704KB - 720896	32KB	Drugi ekran k.graficznej (jego 1 połowa) LUB 8 planów ekranu tekstowego (MONO)
B8000 - 736KB - 753664	32KB	Drugi ekran k.graficznej (jego 2 polowa) LUB 8 planów ekranu tekstowego (COLOR)
C0000 - 768KB - 786432	32KB	BIOS Karty Graficznej
C8000 - 800KB - 819200	160KB	BIOS Shadow Area
F0000 - 960KB - 983040	64KB	BIOS

Tab. State of first megabyte of RAM memory after SSBL stops working.

1.2.3 Stage 3 – Loader

After taking control, loader begins to load kernel. The second stage is to initiate basic RAM Memory Manager structures, necessary to correct root application data loading. These structures will be then used by kernel. When MM is ready, the loader writes INIT.DAT file to memory. This file stores root's application data which should be started. The next stage is to load applications to memory (applications are only loaded, without initiating themselves). The address that programs were loaded into are written in Process initiation array (read from INIT.DAT) for future kernel use. When all these stages are applied, loader switches the processor into Protected Mode and gives control to kernel.

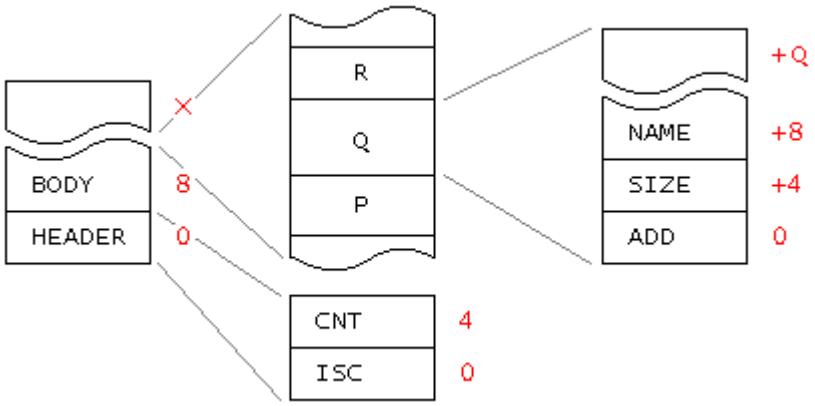
Adres początkowy	Rozmiar	Opis
00000 - 0KB - 000000	1KB	Tablica wskazów przerwań RM
00400 - 1KB - 001024	172B	BDA (Bios Data Area)
004AC - 1196B - 001196	68B	Zarezerwowane przez IBM
004F0 - 1264B - 001264	16B	Obszar komunikacji użytkownika
00500 - 1280B - 001280	2816B	Obszar danych (A)
01000 - 4KB - 004096	12KB	Obszar zmiany kontekstu. (B)
04000 - 16KB - 016384	16KB	Kernel
08000 - 32KB - 032768	4KB	GTB
09000 - 36KB - 036864	4KB	TB (4KB dla do 128MB RAM)
0A000 - 40KB - 040960	4KB	ETB (4KB dla do 128MB RAM)
0B000 - 44KB - 045056	X 572 -X -Y Y	APLIKACJE --- PLIK INICJACJI APLIKACJI
9A000 - 616KB - 630784	8KB	TEMP
9C000 - 624KB - 638976	8KB	Kernel - IOPRBM
9E000 - 632KB - 647168	4KB	Kernel - Stos
9F000 - 636KB - 651264	3KB	R
9FC00 - 639KB - 654366	1KB	EBDA (Extended BIOS Data Area)
A0000 - 640KB - 655360	64KB	Pierwszy ekran k.graficznej
B0000 - 704KB - 720896	32KB	Drugi ekran k.graficznej (jego 1 połowa) LUB 8 planów ekranu tekstowego (MONO)
B8000 - 736KB - 753664	32KB	Drugi ekran k.graficznej (jego 2 polowa) LUB 8 planów ekranu tekstowego (COLOR)
C0000 - 768KB - 786432	32KB	BIOS Karty Graficznej
C8000 - 800KB - 819200	160KB	BIOS Shadow Area
F0000 - 960KB - 983040	64KB	BIOS Systemu (glowny ladowany z ROMu)

Tab. State of first megabyte of RAM memory while kernel loading.

1.2.4 Process initiation array

File INIT.DAT includes "Process initiation array" structure. This array includes all root's application data necessary for loader. All the file may be divided into 2 parts: Headline and table body. Array headline consists of two fields:

ISC Entire INIT.DAT file size, the field uses as checksums which protects system from using damaged INIT.DAT file.
CNT This field stores the entire number of applications to load.



Pic. INIT.DAT file Process initiation array structure.

Array body however consists of CNT variable size fields, and each of them describes one application (shown as "Q" field in the picture):

ADD This field is always cleared in INIT.DAT file, and after loading the application to memory, the field stores its address in RAM.
SIZE This is required application size. The field is useful to both prevent system from loading damaged application and to initiate application by kernel afterwards.
NAME This is string of variable length signs, which is completed with zero byte. It stores application name.

1.2.5 Boot application data configuration

In order to configure own list of applications to load with kernel you should use managing application MAKEINIT. It is simple to make your own INIT.DAT file using MAKEINIT. To do this you should place MAKEINIT in folder where you store applications. Then you should start it from command, treating the applications as MAKEINIT call parameters. Application automatically generates INIT.DAT file, which together with previously pointed applications should be copied to main directory of medium that system boots from (e.g. for FDD it is a:\ folder). If MAKEINIT was called in folder where INIT.DAT exists, then the file is overwritten.

1.3 Memory

From 0.1 version ACD system is equipped with simple memory management mechanism. Because paging mechanism is used to manage process space, kernel memory manager has 4KB allocation granularity. Because of this, bit arrays function as memory management mechanism. Basic bit array TB, and expanded ETB bit array both store memory frames data state. Every 4KB memory frame assigns one bit in TB and one bit in ETB. Position of the bit in arrays assigns frame number described by the one. E.g. frame 0 assign bit 0 in byte 0, whereas frame 34 assign bit 2 in byte 4. The bits sampled from TB and ETB both define pair which describes the frame (therefore possible are 4 frame states).

GTB	TB	ETB	Opis
0	0	0	Wolna ramka gotowa do alokacji.
1	0	1	Wolna ramka ze starymi danymi, czeka na czyszczenie.
1	1	0	NIEALOKOWALNA RAMKA: - ramka używana przez sprzęt - nieistniejąca fizycznie ramka - ramka uszkodzonego obszaru RAM
1	1	1	Ramka zarezerwowana, zaalokowana przez proces

Tab. Bit state relationship in three arrays.

With maximum memory size, TB and ETB store a number of 2^{20} frames data (above one million). Searching such a big array for finding free allocation frame would take too much time (even testing 32 bits simultaneously you should apply, considering you were a pessimist, 32768 comparisons). In order to avoid this, there is the third bit array called Global Bit Array (GTB). Every single bit in GTB describes 32 frames. If there is single free frame, the Bit takes null value, else it takes 1 value. Owing to GTB even 4GB system memory may be enough to check 256 frames simultaneously, which means, in the worst case, 1024 tests made in order to find a free frame.

1.4 Tasks

Every single process in ACD system is represented by (*Process Description Block*). This block include all essential process data, its state, mailbox (see chapter 1.8), and if needed IOPBM copy (see chapter 1.6). Every PDB is 4KB long (or 12KB including IOPBM copy), because of memory management mechanisms used in kernel. (see chapter 1.4).

1.4.1 VPS

(Virtual Process Space)

Owing to apply paging mechanism, every single process has its own address space known as VPS. First 4MB of this space is reserved to mapped from it kernel space (VSS – *Virtual System Space*). From 4MB address process code area begins, and then extendible stack data. At the end of VPS, beginning from 4GB downwards there is process stack.

From version 0.01 kernel share process so much VPS space, owing to applying VSS mapped memory in the ratio of 1 to 1 (physical address = virtual address). Every time you switch to kernel level paging switch-over occurs. Owing to this kernel may access all the system memory (the method, however, is rather slow as it needs two paging switch-over, and from 0.2 version it will be replaced with kernel in full VSS).

1.4.2 Process states

In ACD system process may take one of four states. The basic one is "New". The process takes this one, if function initiating it is called. If all system structures are built, the process is ready to be applied, and it switches to "Ready" status. In this status it waits till executing. In due time, the system allocates it a processor switching it to "Working" status. In ACD system switch-over between the contexts was implemented according to program, because it turned out to be faster from process description segments as well as hardware mechanism. When the time quantum using for process function passed, or system API is called, or any other interrupt occurs, the process is stopped and moved back to "ready" queue (it's because ACD system uses *preemptive multitasking*).

Pic. Four-state process model.

This may be a state that process waits for relevant message. Then it is moved to process queue waiting till the message comes.

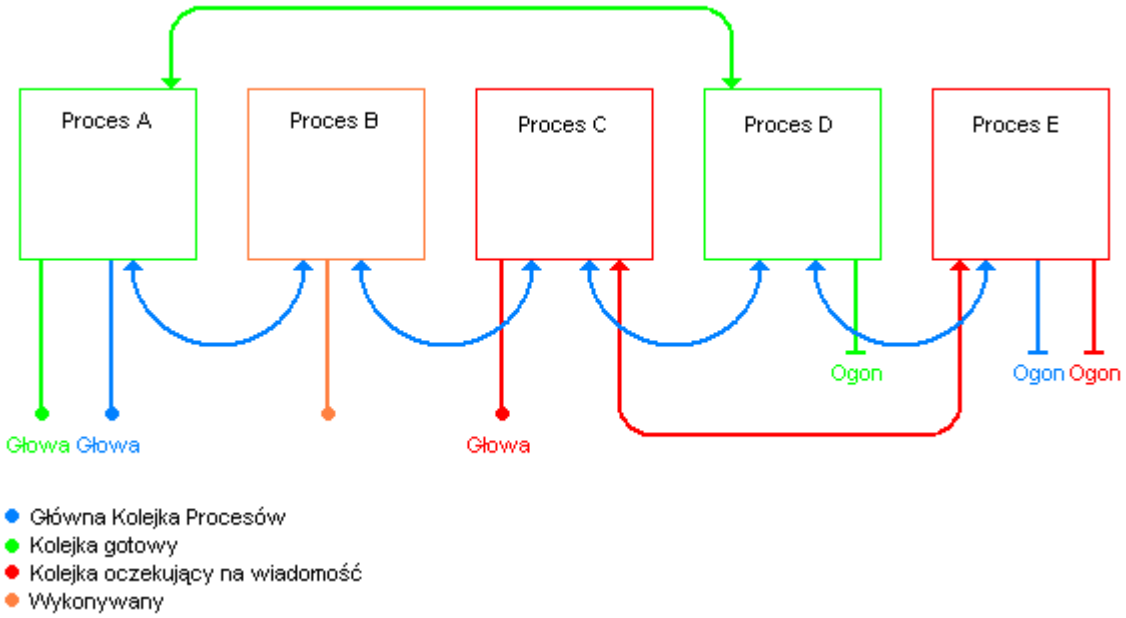
1.4.3 Process Description Block

Offset	Rozmiar	Nazwa	Opis
0	4B	PID	Numer identyfikacyjny procesu.
4	4B	FID	Numer identyfikacyjny procesu rodzicielskiego.
8	4B	UID	Numer identyfikacyjny właściciela.
12	4B	EAX	Stany rejestrów ogólnego przeznaczenia (#GP - General Purpose) podczas wystąpienia przerwania.
16	4B	EBX	
20	4B	ECX	
24	4B	EDX	
28	4B	ESI	
32	4B	EDI	
36	4B	EBP	
40	4B	EIP	Informacje odkładane przez procesor podczas wywołania przerwania i przywracane IRETD.
44	4B	EFLAGS	
48	4B	ESP	
52	4B	R	Zarezerwowane
56	4B	CR3	Adres Katalogu Stron (wskaz do VPS)
60	4B	IOPLBM1	Adres do Bitmapy dostępu portów I/O (część 1)
64	4B	IOPLBM2	Adres do Bitmapy dostępu portów I/O (część 2)
68	4B	GKPU	Poprzedni PDB w głównej kolejce procesów
72	4B	GKPD	Następny PDB w głównej kolejce procesów
76	4B	KSP	Poprzedni PDB w kolejce stanu
80	4B	KSN	Następny PDB w kolejce stanu
84	1B	STAN	Stan w jakim znajduje się aktualnie proces
85	3B	R	Zarezerwowane
88	1B	HV	Wysoka wersja programu
89	1B	LV	Niska wersja programu
90	2B	COMP	Numer kompilacji
92	1B	CBHV	\ Najstarsza wersja z która ten program jest
93	1B	CBLV	> w pełni zgodny wstecz.
94	2B	CBCOMP	/
96	32B	NAZW	Nazwa obrazu (32znakow, zero domyślne)
128	4B	CHSUM	Suma kontrolna dla adresów IPC Zawiera negacje adresu PDB
132	4B	MLBS	Głowa kolejki w skrzynce (IPC 2.0+)
136	4B	MLBE	Ogon kolejki w skrzynce (IPC 2.0+)
140	1B	MCNT	Ilość wiadomości w skrzynce (IPC 2.0+)
141	3B	R	Zarezerwowane
144	4B	MEMBOT	Dolna granica przestrzeni danych procesu
148	4B	MEMTOP	Górna granica przestrzeni danych procesu
152	4B	STABOT	Dolna granica stosu
156	4B	MEMFR	Ilość ramek zajmowanych przez proces (k+d+s)
160	1888B	R	Zarezerwowane
2048	2048B	MES	Skrzynka odbiorcza procesu, przechowuje do 128 wiadomości IPC, każda o rozmiarze 16 bajt (w tym 12 bajt na treść wiadomości i 4 nagłówka).

Tab. Process description block structure (PDB).

1.4.4 Queuing mechanism

Process management takes place by means of bi-directional queues with head and tail pointers. Each process belongs to Main Process Queue. This makes it possible to read all system process condition following the queue. The second queue, that each process belongs to, is condition queue. There may exist a number of condition queues, but the process always belongs to any (this, which queue the process belongs to is indicated by “STAN” field in its PDB).



Pic. Relationship between queues and PDB.

As we can see on the diagram above, there may be many queues from kernel side. However, from PDB side there are always four pointers describing 2 queues. In this case A and D processes are waiting to be executed, forming "Ready" queue, C and E processes form "Suspended" queue, waiting for a message. B process is in "Working" condition.

1.5 Ports

In order to protect ports from unauthorized particular applications access, Secure Mechanism was implemented in ACD system. The process must make a request to allocate him a port. And if system decides, that a definite port can be allocated, it takes the action, and signs it by proper return-from-procedure code. If process tries to read or write at port, that it has not reserved, system will not allow to do this and will terminate it. After terminating an action at port the process should release one, so that the other process might access it. If process does not release ports they will be released only after terminating one. Port management mechanism let you reserve single, double, or quadruple ports. All the actions connected with ports are available in group 3, 0x80 interrupts.

1.5.1 Ports and x86 architecture

x86 family architecture shares device process management mechanism. However it is slower than software mechanism, in practice ACD system creates only one TSS block (*Task State Segment*), which is used to describing all processes. In order to get secure ports, TSS block point at IOPBM (*I/O Permission Bit Map*). IOPBM describes every possible port by means of single bit, which gives 8KB of its size together (it is because in PC computer there may exist 2^{16} ports). Cleared bit means that process may use the port, and set bit means that process may no permission to use the port.

1.5.2 Port's protection implementation

Because ACD system uses one TSS block for all processes, it means that every process change force IOPBM update. The problem was solved by creating private copies of IOPBM array for every process which request the port. It means that process which work on ports get additional block of 8KB that stores information which ports may be used or not. When the process is resumed secure port in TSS is turned on, and its private IOPBM replace previous IOPBM in TSS. The processes not working on ports don't have its own IOPBM copy, because they are created at first request of process to allocate the port. This mechanism prevents from memory excessive wasting, and speeds up switching between process contexts.

The basic aim of secure port is to manage resources simply and safely. To do this, the following rule was established: every port may be used only by one application at once. Owing to this rule kernel introduces basic process working protection on ports and what is most important, enable running more composed user application, which will protect ports regardless of kernel.

To work properly the rule above require introducing mechanism which synchronize all IOPBM copies. Kernel beside single TSS and its IOPBM creates IOPRBM (*I/O Protection Bit Map*). Each require and release of port pass through mechanism which checks the port in IOPRBM. If bit on IOPRM port is cleared, it means that one of processes has just reserved it, and if bit is set the port may be reserved. Therefore IOPRBM is just negation of all process mirror IOPBM copies' rule.

1.6 Interrupts

Management system have not been implemented yet.

1.7 Inter-Process Communication

(IPC - Internal Process Communication)

Inter-Process Communication is shared by means of group 5, 0x80 interrupt. Because IPC reports are a base of modular system, forwarding them must be express. Therefore, the following solutions have been implemented:

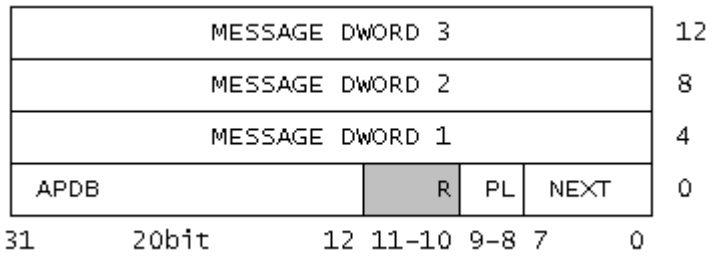
IPC in ACD system is based on direct forwarding a message of the same size. It is very efficient, because the time is not wasted for searching message in shared mailboxes. Each process together with its creation gets its own inbox which may contain a number of 128 messages. Additionally, to achieve maximum capacity message addressing was used. However, not by means of process identity, but their inbox addresses (new 0.02 kernel version mechanisms will allow to address message by process identities).

1.7.1 Priorities

Each message may get significance priority. There exist four preference levels, from zero (the lowest, default) which means regular message, to third (the highest) which means the message with the highest preference level. Messages with higher priorities are delivered faster than messages with lower ones. This means, that a message with priority of level 1 preference may be received before 0 level priority message is delivered to a mailbox. Processes have full control considering priorities they want to allocate to their message.

1.7.2 Message Structure

Inbox of each process has always 2KB (include 128 fields, each has 16bytes). Each field consist of 4 bytes of headline and 12 bytes of message.



Pic. Message structure in mailbox.

APDB PDB sender address.
R Field reserved for future usage.
PL Level of message preference. May be from 0 (regular message) to 3 (message of a highest priority). Message with PL = 3 may be received before three messages previously received to a mailbox, message with PL = 2 before two etc.
NEXT Stores number of field, in which there is next message. Fields are numbered from 0 to 127.

1.7.3 Mailbox management

Incoming messages are organized as one-way list with pointers at its beginning (MLBS field in PDB) and end (MLBE field in PDB).in addition, PDB include MCNT field, which stores number of messages in mailbox. If mailbox is empty, all three fields ale cleared, and if it is full MCNT field takes 128 value. Storing new message bases on linear mailbox search in order to find an empty field, store message and to increment MCNT. Loading message causes copying it to process records, disconnecting it from queue, and clearing the field it used. In the end MCNT counter is decremented.

2 Working with system

2.1 Compilation

Because of modular system structure, possible is running nearly every executable file format. It is possible due to two-stage running an application. The first stage consist in loading executable program file, by optional user-level applications. The application takes proper actions on mirror loaded, so as eg. reallocation. When the mirror is ready, it is forwarded to kernel to be run by API function. Kernel runs only process mirror in binary shape ready for direct processing by a processor. It is just second stage of running an application mechanism.

2.1.1 Compilation In Assembler

In order to compile an application written in assembler properly, you should apply a universal kernel-processed binary code rule mentioned. From 0.01 kernel version, executable process code is installed under 4MB address. Therefore, in order to compile eg. application by means of NASM you should apply the following steps:

- Place the following information for compiler in code:
```
[BITS 32]
[ORG 0x400000]
```

- Compile to BIN format, eg.:
```
C:\>nasm –f bin hello.asm –o hello.bin
```

2.1.2 Compilation In ANSI C

Compilation of C code bases on the same rules that ASM code. Below there is example of compilation by means of GCC:

```
C:\>gcc -c hello.c -o hello.o
C:\>ld --oformat binary hello.o -o hello.bin -Ttext=0x400000
```

2.2 System calls
(API – Application Interface)

ACD system kernel shares basic procedure package to applications. This enable Inter-Process Communication, memory management etc. All these procedures (called also system calls) are available by program calling 0x80 interrupt. Parameters are forwarded to procedures by means general purpose registers. To make working simpler, calls were additionally grouped according to few categories. The number of called function is written in AL record as follows: four more significant bits (hi nibble) is a group number, and four less significant bits (lo nibble) is a number of function in a group. Eg. AL = 0x31 means, that the first function from the third group will be run. Below there is a detailed description of all available calls.

Group 0 – Main calls

No.	Version	Date	Name
0x00	-	-	-
0x01	1.1	04.05.2007	System version
0x02	1.1	04.05.2007	Up time in seconds
0x03			Wait (ms)

Group 1 - Processes

No.	Version	Date	Name
0x10			Find a process
0x11	1.1	13.05.2007	Run a process [NFT]
0x12	1.1	13.05.2007	Kill a process
			Return the rest of time unit

Group 2 – Memory

No.	Version	Date	Name
0x20	2.1	13.05.2007	Memory allocated by a process
0x21	1.1	13.05.2007	Increasing data area [NFT]
0x22	1.1	13.05.2007	Decreasing data area [NFT]
0x23	1.1	13.05.2007	Stack size
0x24			Allocate under address
0x25			Allocate physical block under address
0x26			Forward data area
0x27			Memory area condition

Group 3 - Ports

No.	Version	Date	Name
0x30	1.1	13.05.2007	Release all the ports
0x31	1.2	13.05.2007	Allocate a port
0x32	1.2	13.05.2007	Release a port
0x33	1.2	13.05.2007	Allocate a double port
0x34	1.2	13.05.2007	Release a double port
0x35	1.2	13.05.2007	Allocate a quadruple port
0x36	1.2	13.05.2007	Release a quadruple port

Group 4 – Interrupt

No.	Version	Date	Name
0x40			Mount an interrupt
0x41			Release an interrupt

Group 5 – IPC

No.	Version	Date	Name
0x50	3.1	07.05.2007	Send a message
0x51	3.1	07.05.2007	Receive a message
0x52	3.1	07.05.2007	Receive a message with awaiting
0x53	2.1	07.05.2007	Get a number of message in mailbox

2.2.1 Group 0 – Main call

0x01 – Kernel version

The function returns kernel version of system running.

Input:

Lack

Output:

EAX – Kernel version in the form of:

- The highest byte – High version
- The lowest byte – Low version
- The lowest two bytes – Compilation

0x02 – System up time

The value returned by these functions is kernel up time counted in seconds (time from running the first process to its execution).

Input:

Lack

Output:

EAX – Up time in seconds.

2.2.2 Group 1 – Processes

0x11 – Run a Process

In order to run a descent process, you should load its mirror to random-access memory, equalizing the initial address to 4KB (ALGIN 4KB). The next step is to create process description block (recommended with equalizing the initial address to 4KB) in the following form:

Example in Assembler:

```
process:
address            dd ?
size              dd ?
comp              dw ?
verlo             db ?
verhi             db ?
compatible_comp   dw ?
compatible_verlo  db ?
compatible_verhi  db ?
owner             dd ?
name     TIMES 32 db ?
```

Example in C:

```
struct process {
unsigned long      address;
unsigned long      size;
unsigned short int comp;
unsigned char      verlo;
unsigned char      verhi;
unsigned short int compatible_comp;
unsigned char      compatible_verlo;
unsigned char      compatible_verhi;
unsigned long      owner;
char               name[32];
};
```

The meaning of fields in structure:

```
address           – Initial address of the process mirror to run.
size              - Process mirror size in bytes.
comp              - Compilation number.
verlo             - Low program version.
verhi             - High program version.

Fields which describes the oldest version of this process, with which the
current version is compatible with backwards:

compatible_comp   - Compilation number.
compatible_verlo  - Low program version.
compatible_verhi  - High program version.

owner             - Defines UID of user that we appoint to be the owner
                    of the process if it was created by root process of
                    system. Otherwise the field should be reseted.
name              - Sequence which represent process mirror loaded.
```

Input:

EBX – Address of a structure which describes the process.

Output:

EAX – Result of action:

0 – Action executed successfully.
1 – Wrong structure address.
2 – Not entire structure is in the memory.
3 – Wrong initial process address.
4 – Wrong position of process mirror.
5 – Not enough RAM to run the process.
6 – Too big process mirror.
7 – Not entire mirror process is in the memory.

Steps of executing function:

- Checking if structure address is within the process data area.
- If it is between the pages, it starts checking if both frames are within VPP.
- Copying the structure to kernel virtual memory.
- Verifying the structure:
  - If mirror initial address is leveled to the beginning of the frame.
  - If initial address is within the process data.
  - If final address is within the process data.
  - If there is enough random-access memory to run the process.
- Checking if process mirror is in random-access memory.
  - Checking by examining if page tables and pages are present.
- Creating PDB (Process Description Block).
- Connecting PDB to process main queue.
- Initiating PDB by start data and data from structure.
- Checking and, if necessary, correcting „compatible backwards version" fields.
- Connecting PDB to execution queue.
- Creating VPS.
- Creating process stack.
- Page mapping from parent VPS and uninstalling them.

0x12 – Delete the Process

Deleting the process is possible, provided it was run by the same user, that process calling functions had been. Else is when the process calling functions was run by system or root. Then it can delete optional process.

Input:

EBX – PID of process to delete.

Output:

EAX - Result of action:

0 - Action executed successfully.
1 – There is no such a process.
2 – You are not allowed to delete this process.

2.2.3 Group 2 – Memory

0x20 – Memory used by process

Function returns size of memory used by process, and size of memory used by system structures assigned to it. Both sizes are given in bytes. Freed areas in memory process data space are included.

Input:

Lack

Output:

EAX – Size of memory used by process.
EBX – Size of memory used by system structures assigned to it.

0x21 – Increasing data area

Function increases process data space to set size (which is multiply of 4Kb) by reserving the missing memory. The function take into account collisions with process stack area, and lack of memory.

Input:

EAX – Size, that process data area will be increased. (on 12 low bits there may be only function call number, the other values will be ignored).

Output:

EAX – Initial address of attached data area, or error number:
1 - Not enough random-access memory to apply the action (physical or virtual).

0x22 – Decreasing data area

The function decreases process data space to set size (which is multiply of 4Kb), by freeing definite size of memory from the end of process data area. The function take into account lack of parameters or forwarding size grater than entire size of data area.

Input:

EAX – Size, that process data area will be decreased. (on 12 low bits there may be only function call number, the other values will be ignored).

Output:

EAX – New closing area data address.

0x23 – Stack Size

Function returns size of process stack, which calls the function.

Input:

Lack

Output:

EAX – Stack size in bytes.

2.2.4 Group 3 – Ports

0x30 – Release of all the ports

Function frees all ports reserved by a definite process.

Input:

Lack

Output:

Lack

0x31 – Port reservation

The function task is to reserve every single port for application or, in case the port was busy, signing it by output code.

Output:

SS:ESP – Port number to be reserved (dword)

Input:

EAX - Result of action:
- 0 - Action executed successfully.
- 1 – Port is just reserved.
- 2 – Not enough random-access memory to apply this action.

0x32 – Port release

Function frees every single port reserved by application or, in case the port was free, the function signs it by output code.

Input:

SS:ESP – Port number to be released (dword)

Output:

EAX - Result of action:
- 0 - Action executed successfully.
- 1 – Port has not been reserved or is still not reserved by these applications.

0x33 – Double port reservation

The function task is to reserve double port for application or, in case the port or a part of it was busy, signing it by output code.

Input:

SS:ESP – Port number to be reserved (dword)

Output:

EAX - Result of action:
0 - Action executed successfully.
1 – Wrong Port number.
2 – Port or a part of it is just reserved.
3 – Not enough random-access memory to apply this action.

0x34 – Release of double port

Function frees double port reserved by application or, in case the port or a part of it was free, signs it by output code.

Input:

SS:ESP – Port number to be released (dword)

Output:

EAX - Result of action:
0 - Action executed successfully.
1 – Wrong Port number.
2 – Port was not reserved, was reserved partly or is not reserved by these applications.

0x35 – Quadruple Port reservation

The function task is to reserve quadruple port for application or, in case the port or a part of it was busy, signing it by output code.

Input:

SS:ESP – Port number to be reserved (dword)

Output:

EAX - Result of action:
0 - Action executed successfully.
1 – Wrong Port number.
2 – Port or a part of it is just reserved.
3 – Not enough random-access memory to apply this action.

0x36 – Quadruple Port release

Function frees quadruple port reserved by application or, in case the port or a part of it was free, signs it by output code.

Input:

SS:ESP – Port number to be released (dword)

Output:

EAX - Result of action:
- 0 - Action executed successfully.
- 1 – Wrong Port number.
- 2 – Port was not reserved, was reserved partly or is not reserved by these applications.

2.2.5 Group 4 – Interrupts

These functions have not been implemented yet.

2.2.6 Group 5 – IPC

0x50 – Send a message

Function sends a priority message to a definite addressee. If he expected the message, it is received immediately, ignoring the inbox, otherwise, if the mailbox is not overfull, the message is recorded to it.

Input:

SS:ESP+12 – Third message dword
SS:ESP+8 – Second message dword
SS:ESP+4 – First message dword
SS:ESP – Receiver address, 1,0 bits – message priority

Output:

EAX - Result of action:
0 - Action executed successfully.
1 – Wrong process address.
2 – Receiver inbox is full.

0x51 – Receive a message

Procedure receives a message from mailbox from a definite or optional sender if exists, or returns error sign. While choosing a message to read, the message priorities are taken into account.

Input:

SS:ESP – PID of sender in request (or 0xFFFFFFFF so optional sender)

Output:

SS:ESP+12 – Third message dword
SS:ESP+8 – Second message dword
SS:ESP+4 – First message dword
SS:ESP – PID of sender
EAX - Result of action:
0 - Action executed successfully
1 – No message
2 – No message from expected sender.

0x52 – Receive a message with awaiting

Procedure receives a message from mailbox from a definite or optional sender if exists, or put process to sleep until it comes. While choosing a message to read, the message priorities are taken into account.

Input:

SS:ESP – PID of sender in request (or 0xFFFFFFFF so optional sender)

Output:

SS:ESP+12 – Third message dword
SS:ESP+8 – Second message dword
SS:ESP+4 – First message dword
SS:ESP – PID of sender

Or

SS:ESP – PID of sender in request in case process is put to sleep.

0x53 – Number of messages in mailbox

Checks if there is any message in mailbox and returns its number.

Input:

Lack

Output:

EAX – Number of messages in mailbox (or 0 in case there is no message).