



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

PROJEKT DYPLOMOWY

*Lekki system operacyjny dla komputera z procesorem x86-64 -
dokumentacja techniczna*

Autor:	Piotr Peter, Michał Żelasko
Kierunek:	Informatyka
Typ studiów:	Stacjonarne
Opiekun pracy:	dr inż. Zaborowski Wojciech

Kraków, 2023

Spis treści

1	Wybrane aspekty realizacji	7
1.1	Zasada działania systemu	7
1.1.1	Zarządzanie pamięcią	8
1.1.2	Stan po uruchomieniu	9
1.1.3	Stos systemowy	11
1.1.4	Linia A20	11
1.1.5	Streaming SIMD Extension	11
1.1.6	Stronicowanie	11
1.1.7	Segmentacja pamięci	15
1.1.8	Task State Segment	17
1.1.9	Przerwania systemowe	20
1.1.10	Sterowniki	21
1.2	Wykorzystane rozwiązania technologiczne wraz z uzasadnieniem ich wyboru .	23
1.2.1	Bootloader	23
1.2.2	Format i ładowanie plików wykonywalnych	24
1.2.3	Interfejs graficzny	26
1.3	Istotne mechanizmy i zastosowane algorytmy	26
1.3.1	System plików	26
1.3.2	Zarządzanie procesami	30
2	Funkcje i struktury	31
2.1	headers/clock_driver.h	31
2.1.1	get_clock_ticks	31
2.1.2	handle_clock_interrupt	31
2.2	headers/deque.h	31
2.2.1	deque_create	32
2.2.2	deque_push_left	32
2.2.3	deque_push_right	32
2.2.4	deque_peek_left	32
2.2.5	deque_peek_right	32
2.2.6	deque_pop_left	33
2.2.7	deque_pop_right	33
2.2.8	deque_destory	33
2.2.9	deque_is_empty	33
2.2.10	struct Deque	33
2.3	headers/desktop.h	34
2.3.1	create_desktop	34
2.4	headers/desktop_selected_zone.h	34
2.4.1	create_selected_zone	34
2.5	headers/event.h	34
2.5.1	event_init	34
2.5.2	event_stop_propagation	35
2.5.3	struct Event	35
2.6	headers/gdt.h	35
2.6.1	fill_gdt	35

2.6.2	load_gdt	35
2.6.3	struct Memory_Segment	35
2.6.4	struct GDT_Register	36
2.7	headers/graphics_mode.h	36
2.7.1	set_graphics_mode	36
2.7.2	add_drag_observer	36
2.7.3	add_release_observer	36
2.7.4	remove_drag_observer	37
2.7.5	remove_release_observer	37
2.7.6	render	37
2.7.7	handle_mouse_event	37
2.7.8	get_mouse_x	37
2.7.9	get_mouse_y	37
2.8	headers/icons/cursor.h	38
2.9	headers/icons	38
2.10	headers/icons.h	38
2.10.1	render_icons	38
2.10.2	select_icons	38
2.10.3	create_icons	38
2.10.4	stop_dragging_icons	39
2.11	headers/image.h	39
2.11.1	image_create	39
2.11.2	image_destory	39
2.12	headers/interrupts.h	39
2.12.1	fill_idt	39
2.12.2	activate_idt	39
2.12.3	struct Gate_Descriptor	40
2.12.4	struct IDT_Pointer	40
2.12.5	struct InterruptDescriptor64	40
2.13	headers/interrupt_handlers.h	40
2.13.1	handle_exception0xnn	40
2.13.2	ignore_interrupt	41
2.14	headers/keyboard_driver.h	41
2.14.1	handle_keyboard_interrupt	41
2.14.2	activate_keyboard	41
2.15	headers/line_A20.h	41
2.15.1	check_A20	41
2.16	headers/list.h	41
2.16.1	list_create	41
2.16.2	list_append	41
2.16.3	list_pop	42
2.16.4	list_insert	42
2.16.5	list_get	42
2.16.6	list_remove	42
2.16.7	list_destroy	43
2.16.8	list_find_index	43
2.16.9	struct List	43
2.17	headers/long_mode.h	43

2.17.1	detect_long_mode	43
2.18	headers/memory.h	43
2.18.1	memcpy	43
2.18.2	init_memory_blocks	44
2.18.3	malloc	44
2.18.4	free	44
2.18.5	bitmask_memcpy	44
2.19	headers/mouse_driver.h	44
2.19.1	activate_mouse	44
2.19.2	handle_mouse_interrupt	45
2.19.3	enum MouseEventType	45
2.19.4	struct MouseEvent	45
2.20	headers/paging.h	45
2.20.1	setup_paging	45
2.21	headers/port.h	45
2.21.1	port_write8	45
2.21.2	port_write16	46
2.21.3	port_write32	46
2.21.4	port_slow_write8	46
2.21.5	port_slow_write16	46
2.21.6	port_slow_write32	47
2.21.7	port_read8	47
2.21.8	port_read16	47
2.21.9	port_read32	47
2.22	headers/synchronized_queue.h	47
2.22.1	synchronized_queue_create	47
2.22.2	synchronized_queue_put	48
2.22.3	synchronized_queue_get	48
2.22.4	synchronized_queue_is_empty	48
2.22.5	synchronized_queue_peek_last	48
2.22.6	synchronized_queue_destory	48
2.22.7	struct SynchronizedQueue	49
2.23	headers/tss.h	49
2.23.1	fill_tss	49
2.23.2	struct tss	49
2.24	headers/types.h	50
2.25	headers/utils.h	50
2.25.1	printf	50
2.25.2	printfHex	50
2.25.3	printfHex16	50
2.25.4	printfHex32	50
2.25.5	printfHex64	51
2.25.6	printf_memory	51
2.25.7	clearScreen	51
2.25.8	min	51
2.25.9	max	51
2.25.10	round	52
2.25.11	in_range	52

2.26	headers/widget.h	52
2.26.1	init_widget	52
2.26.2	widget_destroy	52
2.26.3	widget_render	52
2.26.4	widget_on_mouse_down	53
2.26.5	widget_on_mouse_enter	53
2.26.6	widget_on_mouse_leave	53
2.26.7	widget_on_mouse_up	53
2.26.8	widget_set_width	53
2.26.9	widget_set_height	54
2.26.10	widget_set_top	54
2.26.11	widget_clear_top	54
2.26.12	widget_set_bottom	54
2.26.13	widget_clear_bottom	54
2.26.14	widget_set_left	55
2.26.15	widget_clear_left	55
2.26.16	widget_set_right	55
2.26.17	widget_clear_right	55
2.26.18	widget_append_child	55
2.26.19	enum WidgetPositionType	56
2.26.20	enum WidgetSizeType	56
2.26.21	struct WidgetStyle	56
2.26.22	struct Widget	57
2.27	headers/windows.h	57
2.27.1	init_windows	57
2.27.2	create_window	58
2.28	headers/windows_widget.h	58
2.28.1	create _w indow	58
2.29	lib/file_system_lib.h	58
2.30	Funkcje	58
2.30.1	callLibrary	58
2.31	lib/multiTasking.h	58
2.32	Funkcje	58
2.32.1	mainFunction	58
2.32.2	newFunction	59
2.32.3	createTask	59
2.32.4	initTasking	59
2.32.5	addTask	59
2.32.6	removeTask	59
2.32.7	yield	59
2.32.8	struct Registers	60
2.32.9	struct Task	60

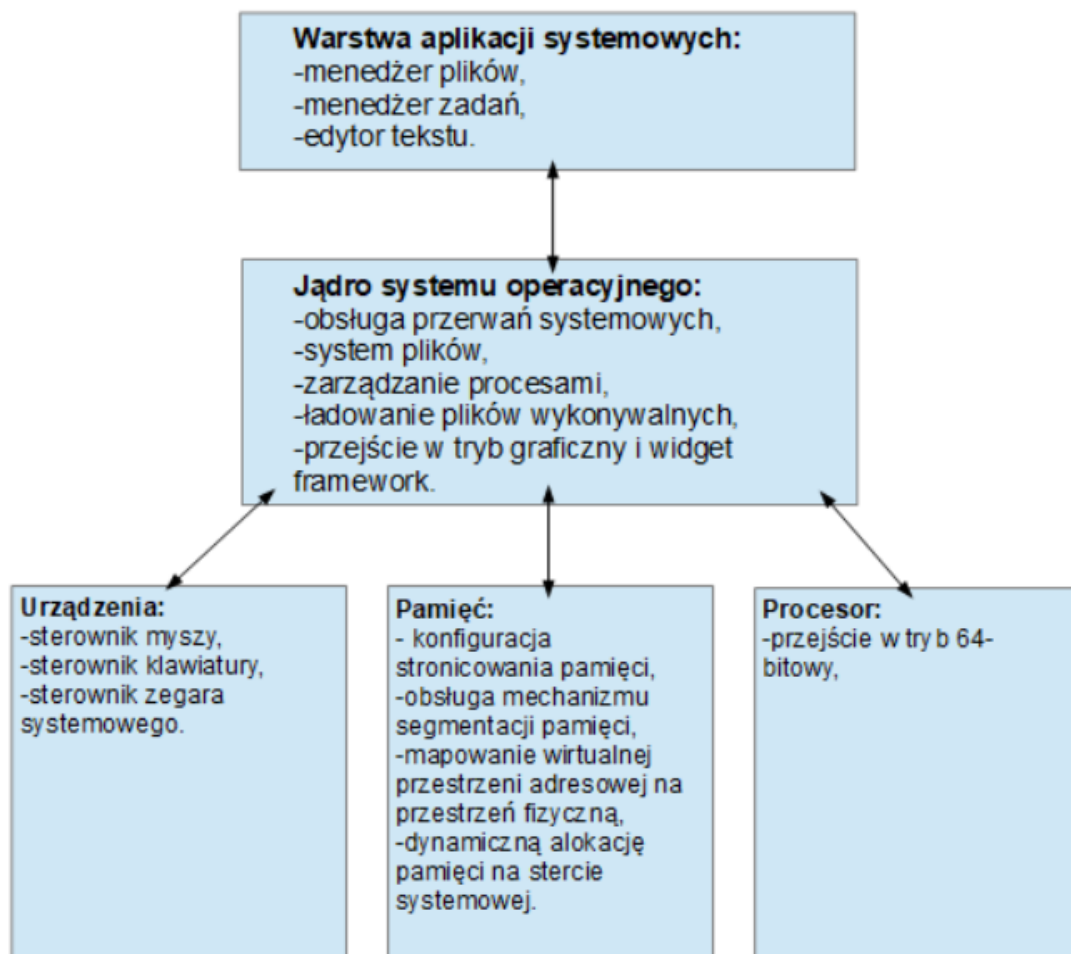
1. Wybrane aspekty realizacji

W ramach rozwoju systemu operacyjnego wyróżniono następujące zagadnienia oraz aspekty wymagające przeanalizowania, zaprojektowania oraz realizacji:

- Uruchomienie jądra systemu za pomocą bootloader'a „Grub”.
- Konfigurację stronicowania pamięci.
- Moduł obsługę mechanizmu segmentacji pamięci.
- Moduł zarządzania pamięcią - wykorzystanie pamięci wirtualnej, mapowanie wirtualnej przestrzeni adresowej na fizyczną.
- Dynamiczną alokację pamięci na stercie systemowej.
- Moduł obsługi przerwań systemowych.
- Moduł obsługi urządzeń wejścia/wyjścia - sterowniki: myszy, klawiatury oraz zegara systemowego.
- Przejście w tryb graficzny.
- Stworzenie hierarchicznej struktury dla wyświetlanych elementów graficznych – prosty gui/widget framework.
- Stworzenie przykładowego oprogramowania systemowego - edytora tekstu dla użytkownika.
- Stworzenie uproszczonej wersji systemu plików.
- Wybór wykorzystywanego formatu plików wykonalnych oraz opracowanie ich oprogramowania do ich ładowania do pamięci.
- Zarządzanie wykonywanymi procesami.

1.1. Zasada działania systemu

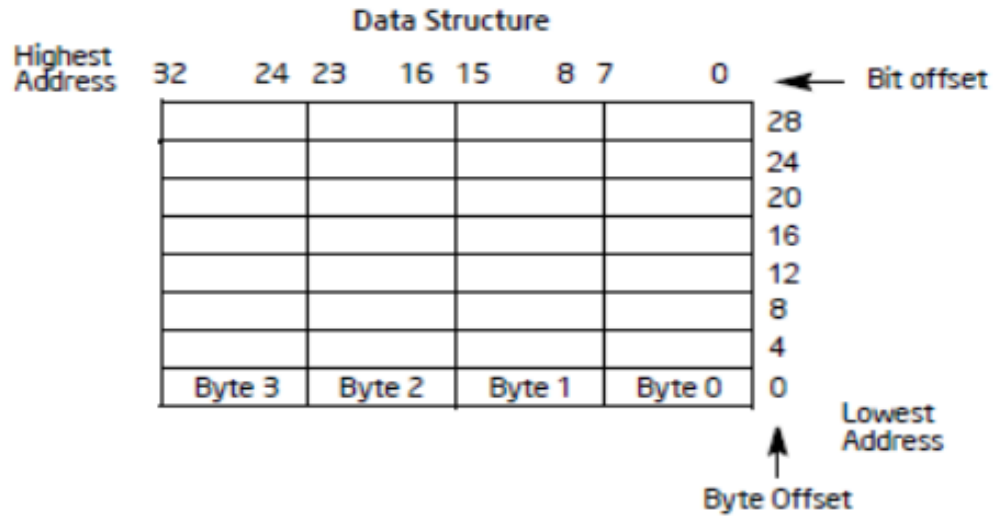
System operacyjny zbudowany jest według przyjętego, ogólnego schematu znanego z różnych wersji Windows czy systemów typu UNIX. Wskazane zagadnienia i aspekty realizacji przyporządkowano do poszczególnych części składowych systemu operacyjnego zgodnie ze schematem przedstawionym na Rys. 1. W takim schemacie architekturę systemu operacyjnego dzielimy na 3 warstwy: pierwszą bezpośrednio kontaktującą się ze sprzętem podzieloną na 3 części odpowiedzialne za kontakt jądra systemu operacyjnego z procesorem, pamięcią i urządzeniami współpracującymi z komputerem (np.: klawiatura), drugą stanowiącą jądro systemu operacyjnego (ang. kernel) i warstwę najwyższą, tworzoną przez aplikacje, z którymi bezpośredni kontakt ma użytkownik (zazwyczaj w trybie graficznym).



Rysunek 1: Wykorzystana notacja.

1.1.1. Zarządzanie pamięcią

Na zamieszczonych w dokumentacji ilustracjach struktur danych umieszczonych w pamięci operacyjnej obowiązuje notacja przedstawiona na Rys. 2. Adresy rosną w górę tabeli. Dla poszczególnych bajtów bity rozmieszczone są od prawej do lewej, od najmniej znaczącego do najbardziej znaczącego. Taka reprezentacja ułatwia interpretację danych, które w pamięci zostaną umieszczone w formacie little-endian, wykorzystywanym na procesorach typu x86-64.



Rysunek 2: Wykorzystana notacja. [?]

1.1.2. Stan po uruchomieniu

Po załadowaniu systemu przez bootloader, procesor powinien pracować w trybie 32 bitowym a rejestry powinny przechowywać informacje zawarte w tabeli 1).

Tabela 1: Zawartość rejestrów po uruchomieniu systemu. [?] [?]

numer bajtu	pole	komentarz
0	flags	(required)
	+-----+	
4	mem_lower	(present if flags[0] is set)
8	mem_upper	(present if flags[0] is set)
	+-----+	
12	boot_device	(present if flags[1] is set)
	+-----+	
16	cmdline	(present if flags[2] is set)
	+-----+	
20	mods_count	(present if flags[3] is set)
24	mods_addr	(present if flags[3] is set)
	+-----+	
28 - 40	syms	(present if flags[4] or flags[5] is set)
	+-----+	
44	mmap_length	(present if flags[6] is set)
48	mmap_addr	(present if flags[6] is set)
	+-----+	
52	drives_length	(present if flags[7] is set)
56	drives_addr	(present if flags[7] is set)
	+-----+	
60	config_table	(present if flags[8] is set)
	+-----+	
64	boot_loader_name	(present if flags[9] is set)
	+-----+	
68	apm_table	(present if flags[10] is set)
	+-----+	
72	vbe_control_info	(present if flags[11] is set)
76	vbe_mode_info	
80	vbe_mode	
82	vbe_interface_seg	
84	vbe_interface_off	
86	vbe_interface_len	
	+-----+	
88	framebuffer_addr	(present if flags[12] is set)
96	framebuffer_pitch	
100	framebuffer_width	
104	framebuffer_height	
108	framebuffer_bpp	
109	framebuffer_type	
110-115	color_info	
	+-----+	

- Eax – zawiera magiczną wartość 0x2BADB002.
- Ebx – zawiera wskaźnik do specjalnej struktury multiboot dostarczonej przez bootloader.

Z powyższej struktury odczytywany jest adres bufora dla trybu graficznego (pole `framebuffer_addr`).

1.1.3. Stos systemowy

Po uruchomieniu systemu rejestr *Esp* może posiadać dowolną wartość. Pierwszym krokiem wykonywanym przez system jest załadowanie do rejestru *Esp* adresu pamięci przewidzianego dla 20 MB stosu systemowego.

1.1.4. Linia A20

Linia A20 jest fizycznym połączeniem, które odpowiada za wartość dwudziestego pierwszego bitu adresu przy każdym odczycie pamięci. Programy pisane na procesory posiadające jedynie 20 linii adresowych (A0 – A19) często wykorzystywały fakt, iż adresy powyżej 1MB, w rzeczywistości powracają do adresu 0. Wprowadzenie linii adresowej A20 pozwoliłoby na uzyskanie dostępu do pamięci powyżej 1MB, co z kolei spowodowałoby błędy w powyższych programach. Aby pozostać kompatybilnym z poprzednimi wersjami procesorów x86 linia A20 jest domyślnie wyłączona – dwudziesty pierwszy bit adresu ma zawsze wartość 0, przez co można uzyskać dostęp jedynie do parzystych megabajtów pamięci.

Po ustawieniu stosu, system wykonuje test sprawdzający czy linia A20 została włączona przez bootloader lub UEFI bądź BIOS. Do dowolnego adresu w pamięci ładowana jest pewna liczba. Następnie do adresu powiększonego o 1MB ładowana jest dowolna inna liczba. W kolejnym kroku porównywane są wartości w obu adresach. Jeśli linia A20 jest włączona to wartości powinny być różne. W przypadku, w którym linia A20 jest wyłączona, adres powiększony o 1MB fizycznie będzie wskazywał na to samo miejsce w pamięci, a zatem obie wartości będą identyczne.

1.1.5. Streaming SIMD Extension

Po sprawdzeniu linii A20 system operacyjny przechodzi do włączenia zestawu instrukcji SSE, pozwalających na wykonywanie obliczeń na liczbach zmiennoprzecinkowych. Włączenie polega na ustawieniu odpowiednich bitów w rejestrach kontrolnych `cr0` oraz `cr4`.

System operacyjny wykonany w ramach pracy inżynierskiej bezpośrednio nie wykorzystuje instrukcji SSE, jednak wykorzystuje wprowadzone wraz z SSE 128-bitowe rejestry do implementacji szybkiego kopiowania pamięci. Przyspiesza to znacząco proces wyświetlania obrazu, podczas którego wykonuje się wiele kopiowań sporych ilości pamięci.

1.1.6. Stronicowanie

W kolejnych krokach system przełącza się w tryb 64 bitowy. Jednak wcześniej wymagana jest konfiguracja pamięci wirtualnej oraz stronicowania pamięci. Od tego momentu procesor będzie operował na adresach wirtualnych, które przy każdym dostępie do pamięci będą tłumaczone na adresy fizyczne. Pamięć fizyczna zostaje podzielona na 4KB fragmenty, dalej określane jako „strony”. Każda strona jest identyfikowana przez swój indeks, tzn. strona zajmująca pamięć od adresów fizycznych 0 – 4KB ma indeks 0, strona od 4KB do 8KB indeks 1 itd.

W pamięci operacyjnej zostają umieszczone „katalogi stron”, czyli struktury danych które zawierają informacje w jaki sposób ma zostać wykonane tłumaczenie adresu, oraz pozwalają na przypisanie praw dostępu do każdej ze stron. Katalogi stron tworzą cztero-poziomową hierarchiczną strukturę, gdzie każdy katalog wskazuje adres kolejnego katalogu leżącego niżej w hierarchii, z którego dane należy odczytać w celu translacji adresu. Katalog leżący najniżej w hierarchii wskazuje już bezpośrednio na indeks strony. Katalogi w kolejności zgodnej z hierarchią to:

- page map level 4 (PML4),
- page directory pointer table (PDPT),
- page directory (PD)
- page table (PT)

Każdy katalog to tablica zawierająca 512 wpisów o rozmiarze 8 bajtów. Wpisy do tabel posiadają następującą strukturę. Aby procesor mógł wykonać tłumaczenie musi posiadać informację o adresie fizycznym tabeli leżącej najwyżej w hierarchii (PML4), adres tej tabeli zostaje umieszczony w rejestrze kontrolnym CR3.

Formaty wpisów przedstawione na Rys. 3 - 7 zostały skonfigurowane w taki sposób, aby tożsamościowo mapować pierwsze 8 GB pamięci.

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page-directory-pointer table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 512-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 512-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page-directory-pointer table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored

Rysunek 3: Format wpisu do tabeli PML4 cz. 1. [?]

Bit Position(s)	Contents
7 (PS)	Reserved (must be 0)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
M-1:12	Physical address of 4-KByte aligned page-directory-pointer table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 512-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Rysunek 4: Format wpisu do tabeli PML4 cz. 2. [?]

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
1 (R/W)	Read/write; if 0, writes may not be allowed to the 1-GByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 1-GByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 1-GByte page; see Table 4-16)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M-1):12	Physical address of 4-KByte aligned page directory referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 1-GByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Rysunek 5: Format wpisu do tabeli PDPT. [?]

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 2-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 2-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	Page size; must be 0 (otherwise, this entry maps a 2-MByte page; see Table 4-18)
10:8	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M-1):12	Physical address of 4-KByte aligned page table referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 2-MByte region controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Rysunek 6: Format wpisu do tabeli PD. [?]

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M-1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 4.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Rysunek 7: Format wpisu do tabeli PT. [?]

1.1.7. Segmentacja pamięci

Procesory x86 wspierają segmentację pamięci, czyli podział przestrzeni adresowej na wiele niezależnych przestrzeni adresowych nazywanych segmentami. Na przykład kod programu i stos mogą być przechowywane w różnych segmentach. Adres instrukcji odnosiłby się zawsze do przestrzeni kodu, a adres stosu do przestrzeni stosu. Następująca notacja jest używana aby odnieść się do bajtu wewnątrz segmentu.

Rejestr segmentu-Adres bajtu

Dla przykładu poniższy adres odnosi się do bajtu AA23 w segmencie na które wskazuje rejestr SS.

SS:AA23

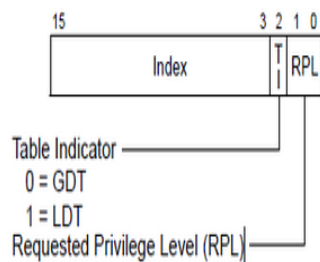
Adres aktualnej instrukcji wykonywanej przez procesor jest identyfikowany przez poniższy adres. Rejestr CS wskazuje na segment kodu, natomiast rejestr EIP zawiera adres instrukcji.

CS:EIP

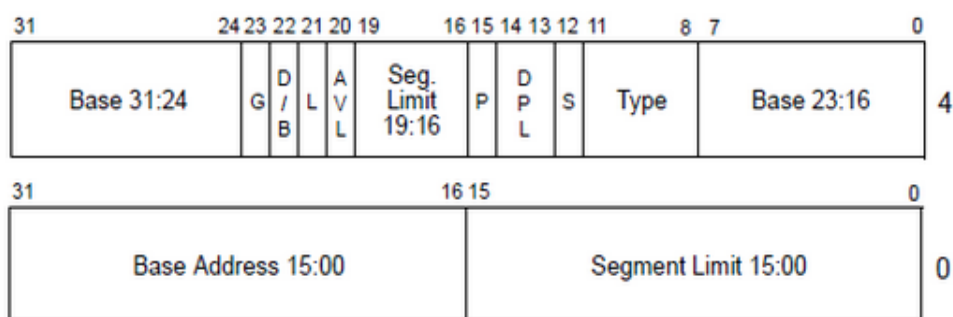
W trybie 32-bitowym segmentacja pamięci jest głównym mechanizmem ochrony dostępu do pamięci. Segmentacja była również niezbędna, aby uzyskać dostęp do całej przestrzeni adresowej, 32-bitowe rejestry pozwalały bowiem na zaadresowanie jedynie 4GB. W trybie 64-bitowym rola segmentacji pamięci została znacznie pomniejszona. Za ochronę dostępu do pamięci odpowiada bowiem mechanizm stronicowania, natomiast segmenty na które wskazują rejestry CS, DS, ES, SS są traktowane tak jakby adres bazowy każdego segmentu był równy 0, niezależnie od wartości przechowywanej w rejestrach. Wyjątkiem są tutaj segmenty wskazywane przez segmenty GS oraz FS, które system operacyjny może użyć do adresowania swoich struktur. W trybie 64-bitowym procesor nie wykonuje także sprawdzenia czy podczas dostępu do pamięci wykroczonego poza zakres segmentu. Sprawia to iż segmenty kodu, stosu i danych zajmują realnie całą przestrzeń adresową, którą ze sobą współdzielą. Pomimo faktu iż segmentacja pamięci jest niemal całkowicie wyłączona w trybie 64-bitowym procesor dalej wykona sprawdzenia poprawności wartości ładowanych do rejestrów segmentów.

Poszczególne segmenty zdefiniowane są w strukturze procesora nazywanej „Global Descriptor Table”, która jest definiowana przez system operacyjny i umieszczana w pamięci RAM. GDT ma format tablicy, której poszczególne wpisy nazywane są deskryptorami segmentu. Rejestry segmentów procesora przechowują 16-bitowe wartości, w formacie widocznym na rysunku 8. Pole RPL wskazuje poziom uprzywilejowania segmentu w skali od 0 do 3, gdzie 0 jest segmentem najbardziej uprzywilejowanym. Do takiego segmentu nie da się uzyskać dostępu z poziomu użytkownika. Mechanizm nadawania poziomu uprzywilejowania dla segmentu, nie jest wykorzystywany przez system i każdy istniejący segment posiada poziom uprzywilejowania 0. Bit 2 wskazuje z której tablicy należy odczytać deskryptor segmentu, oprócz GDT, zdefiniowana jest także tabela „Local Descriptor Table”, która nie jest przez system wykorzystywana zatem pole to zawsze przyjmuje wartość 0. Bity 3-15 to indeks deskryptora z GDT.

Wpisy w tabeli GDT posiadają rozmiar 8 bajtów i mają format opisany na rysunku 9.



Rysunek 8: Format selektora segmentu. [?]



- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

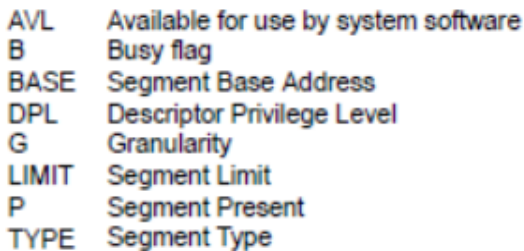
Rysunek 9: Format deskryptora segmentu. [?]

Pierwszy wpis do tablicy GDT nie jest używany przez procesor i powinien być wypełniony zerami. Selektory segmentu, które wskazują na ten wpis noszą nazwę „Null segment selector”. Procesor nie generuje wyjątku, kiedy do rejestru segmentu (innego niż CS i SS) ładujemy „null segment selector”. Wyjątek jest jednak generowany w przypadku próby dostępu do takiego segmentu. „Null segment selector” może być wykorzystany do zainicjowania nieużywanych rejestrów segmentów.

W systemie w GDT zostały umieszczone następujące deskryptory (podane wartości heksadecymalne po umieszczeniu w pamięci zostaną zamienione na little-endian, w ten sposób łatwiej jest dopasować poszczególne bity do tabeli 4):

- Null Descriptor: 0x0000000000000000.
- Deskryptor segmentu kodu: 0x00209A0000000000.


- Deskryptor ten opisuje specjalny segment systemowy i zajmuje rozmiar 2 normalnych wpisów do tabeli GDT. Wpis jest w formacie widocznym na Rys. 10.



Informacje potrzebne do przywrócenia zadania po wystąpieniu przerwania są przechowywane w specjalnym segmencie systemowym nazywanym task-state segment (TSS). W systemie TSS został wykorzystany do skonfigurowania osobnego stosu dla przerw systemowych (mechanizm omówiony w rozdziale 11). Konfiguracja osobnego stosu przerw okazała się niezbędna, ponieważ używany kompilator języka C (gcc) generuje kod, w którym wartość rejestru RSP nie zawsze odpowiada wierzchołkowi stosu. W efekcie po wystąpieniu przerwania część danych na stosie była nadpisywana przez procesor.

W segmencie tym wszystkie pola zostały ustawione na 0, poza IST1-7, które otrzymały adres stosu przerwań (widoczny na Rys. 11).

31	15	0
I/O Map Base Address	Reserved	100
Reserved		96
Reserved		92
IST7 (upper 32 bits)		88
IST7 (lower 32 bits)		84
IST6 (upper 32 bits)		80
IST6 (lower 32 bits)		76
IST5 (upper 32 bits)		72
IST5 (lower 32 bits)		68
IST4 (upper 32 bits)		64
IST4 (lower 32 bits)		60
IST3 (upper 32 bits)		56
IST3 (lower 32 bits)		52
IST2 (upper 32 bits)		48
IST2 (lower 32 bits)		44
IST1 (upper 32 bits)		40
IST1 (lower 32 bits)		36
Reserved		32
Reserved		28
RSP2 (upper 32 bits)		24
RSP2 (lower 32 bits)		20
RSP1 (upper 32 bits)		16
RSP1 (lower 32 bits)		12
RSP0 (upper 32 bits)		8
RSP0 (lower 32 bits)		4
Reserved		0

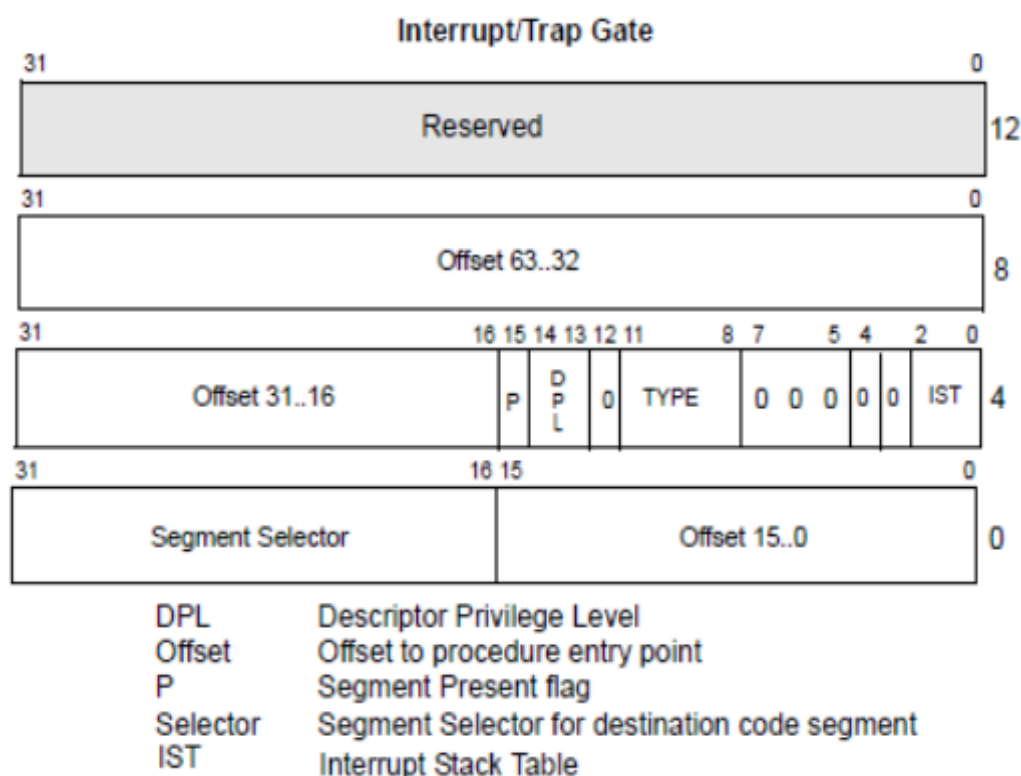
 Reserved bits. Set to 0.

Rysunek 11: Format TSS w trybie 64-bitowym. [?]

1.1.9. Przerwania systemowe

Procesor zapewnia dwa mechanizmy przerwania wykonywania aktualnego zadania. Są to przerwy i wyjątki. Wyjątki występują po wykonaniu instrukcji, jeśli procesor znajdzie się po niej w niewłaściwym stanie. Przerwy natomiast wywoływane są przez urządzenia, które chcą komunikować się z procesorem, lub specjalną instrukcją procesora. Aby przerwy mogły występować konieczne jest ustawienie flagi przerwań w rejestrze flag procesora. Obsługa przerwań i wyjątków odbywa się w niemal jednakowy sposób. Każdy wyjątek/przerwanie ma przypisany swój unikatowy numer. Na jego podstawie po wystąpieniu przerwania procesor odczytuje odpowiedni wpis ze specjalnej struktury umieszczonej w pamięci operacyjnej „Interrupt Descriptor Table” (IDT). Wpis zawiera adres funkcji, która następnie zostanie przez procesor wywołana w celu obsłużenia przerwania. Po wykonaniu funkcji, procesor wznowia wykonywanie kodu od instrukcji, przy której wystąpiło przerwanie.

Format wpisu do tabeli IDT na procesorach x86-64 widoczny jest na poniższej grafice (Rys. 12). Wpisy IDT nazywane są bramkami. Za pomocą pola IST wybierany jest jeden z 7 stosów z TSS, który będzie wykorzystywany podczas przerwania (wartość 0 oznacza pozostanie przy obecnym stosie). Pole TYPE pozwala wybrać typ bramy jako „interrupt gate” lub „trap gate”. Jeśli przerwanie odwołuje się do „interrupt gate” procesor automatycznie wyzeruje flagę przerwań, przez co w trakcie obsługi przerwania nie wystąpi kolejne przerwanie. W przypadku „trap gate” wartość flagi przerwań nie zostaje zmieniona. Jest to jedyna różnica między tymi bramkami. W momencie wystąpienia przerwania rejestry SS, RSP, RFLAGS, CS oraz RIP są odkładane na stosie, w takiej kolejności. W przypadku wystąpienia wyjątku po rejestrach na stosie odkładany jest także kod błędu. Jest to jedyna różnica w obsłudze przerwań i wyjątków.



Rysunek 12: Format wpisu do tabeli IDT. [?]

Stworzony system operacyjny implementuje przerwy w następujący sposób. Tabela IDT posiada 256 wpisów, obsługujących wszystkie możliwe przerwy. Wywołana funkcja odkłada

na stosie kluczowe rejestry, których wartość mogłaby zostać zmieniona w trakcie obsługi przerwania. Mogłoby to spowodować błędy po powrocie do wykonywania kodu, w trakcie którego wystąpiło przerwanie. Po odłożeniu rejestrów na stosie wywoływana jest funkcja napisana w języku C, która jako argument przyjmuje numer przerwania. W dalszej kolejności na podstawie numeru przerwania jest ono ignorowane lub wywoływany jest odpowiedni fragment sterownika (program służący do komunikacji ze sprzętem) urządzenia, które spowodowało przerwanie. Na końcu przywracane są wartości rejestrów odłożonych na stosie i wywoływana jest instrukcja *iret*, która pozwala procesorowi zakończyć obsługę przerwania.

1.1.10. Sterowniki

Sterowniki są programami które służą do komunikacji ze sprzętem. W ramach systemu operacyjnego zrealizowane są sterowniki myszki, klawiatury oraz zegara.

Wymiana informacji z urządzeniami odbywa się za pomocą tzw. portów I/O. Porty są adresowalne za pomocą ich numeru. Znając numer portu, istnieją instrukcje procesora pozwalające na zapis lub odczytanie danych. Istnieje 216 8-bitowych portów. Każde dwa kolejne porty 8-bitowe mogą być traktowane jako port 16-bitowy oraz każde cztery kolejne porty 8-bitowe mogą być traktowane jako port 32-bitowy.

W stworzonym systemie operacyjnym zostały zaimplementowane sterowniki klawiatury, myszy oraz zegara systemowego. Zostały one zaprojektowane tak, aby podczas inicjalizacji dokonywały aktywacji sprzętu, a następnie zajmowały się jedynie obsługą przerw systemowych.

Myszka wraz z klawiaturą komunikują się z procesorem za pośrednictwem kontrolera PS/2. Kontroler posiada swój własny stan, który przechowywany jest w jego pamięci w dwóch bajtowych rejestrach (w formacie widocznym na Rys. 13 i w tabeli 2). Wyróżniamy bajt poleceń oraz bajt statusu. Kontroler na podstawie zawartości rejestrów może wyłączyć generowane przez mysz i klawiaturę przerwanie systemowe lub całkowicie uniemożliwić komunikację z tymi urządzeniami przez porty IO. Z tego powodu istotne jest ustawienie obu bajtów kontrolera na odpowiednie wartości przy inicjalizacji sterowników.

	MSb				LSb			
AT-compatible mode:	--	XLAT	PC	_EN	OVR	SYS	--	INT
PS/2-compatible mode:	--	XLAT	_EN2	_EN	--	SYS	INT2	INT

Rysunek 13: Bajt poleceń kontrolera PS/2. [?]

Tabela 2: Struktura stanu kontrolera PS/2.

Bit	Znaczenie	Komentarz
0	Output buffer status (0 for empty and 1 as full)	Musi być ustawiony przed próbą odczytania danych z IO
1	Input buffer status (0 for empty and 1 as full)	Musi być ustawiony przed próbą wypisania danych na IO
2	Flaga systemowa	Bit ustawiany jeśli system operacyjny przechodzi własne testy
3	Komenda lub dane	0 - dane dla PS/2 device, 1 - dane dla PS/2 controller command
4	Nieokreślone (zależne od chipset)	Częściej używane na nowszych systemach operacyjnych
5	Nieokreślone (zależne od chipset)	receive time-out lub pełny second PS/2 port output buffer
6	Time-out	0 - brak błędu, 1 - błąd
7	Błąd parytetu	0 - brak błędu, 1 - błąd

Istotne z punktu tworzenia sterowników pola bajtu poleceń to:

- INT – ustawienie bitu powoduje generowanie przerwań przez klawiaturę,
- INT2 – ustawienie bitu powoduje generowanie przerwań przez myszkę,
- _EN – wyzerowanie bitu daje możliwość komunikacji z klawiaturą,
- _EN2 – wyzerowanie bitu daje możliwość komunikacji z myszką.

Do komunikacji z kontrolerem przeznaczone są dwa porty IO, są to:

- port komend 0x64 – służy do przesyłania poleceń,
- port danych 0x60 – służy do przesyłania wszelkich innych informacji,

Inicjalizacja klawiatury polega ustawieniu bitu nr 1 bajtu poleceń, jest to wymagane przez kontroler PS/2 aby mógł wysyłać przerwania systemowe. Polega to na przesłaniu do portu komend, bajtu 0xae. Poza aktywacją sprzętu sterownik obsługuje także przerwania systemowe. W trakcie przerwania odczytywany jest bajt z portu danych, który identyfikuje wciśnięty na klawiaturze przycisk. Informacja o wciśniętym przycisku umieszczana jest w kolejce systemowej.

Kolejnym sterownikiem jest sterownik myszki. Do komunikacji z myszką używane są te same porty IO co do komunikacji z klawiaturą. Synchronizacja dostępu do portów nie sprawia jednak żadnych problemów – każdy sterownik korzysta z portów w trakcie swojego przerwania systemowego. Bramki przerwań dla myszy i klawiatury ustawione są jako *interrupt gates*, przez co oba przerwania nie mogą wystąpić jednocześnie.

Inicjalizacja myszki polega, podobnie jak w przypadku klawiatury na ustawieniu odpowiednich bitów w bajcie poleceń kontrolera PS/2. Następnie do portu IO 0x64 należy wysłać bajt 0xF4, po czym myszka zaczyna wysyłać przerwania. Przy uruchomieniu systemu operacyjnego na prawdziwym sprzęcie (nie maszynie wirtualnej) niezbędne przy komunikacji z myszką okazało się oczekiwanie podczas zapisu bądź odczytu danych z kontrolera PS/2. Dane mogą zostać przesłane do myszki jedynie gdy bit nr 1 statusu kontrolera jest wyzerowany. Odczyt danych może nastąpić jedynie gdy bit nr 0 statusu kontrolera jest ustawiony.

Sterownik myszki obsługuje przerwania i umieszcza informację o przesunięciach myszki w kolejce systemowej.

Ostatnim sterownikiem jest sterownik zegara systemowego. Zegar nie wymaga inicjalizacji, generuje przerwania systemowe od razu po ich włączeniu. Sterownik zlicza jedynie ilość obsłużonych przerwań. Ilość tyknięć zegara udostępniana jest dla innych modułów i może posłużyć do odmierzenia czasu. Każde tyknięcie zegara powoduje ponowne renderowanie ekranu.

1.2. Wykorzystane rozwiązania technologiczne wraz z uzasadnieniem ich wyboru

1.2.1. Bootloader

System operacyjny jest zgodny ze specyfikacją multiboot, która opisuje w jaki sposób bootlo-ader może ładować system operacyjny na procesory typu x86. Do uruchomienia systemu operacyjnego wykorzystywany jest bootloader Grub, zazwyczaj używany do uruchomienia systemu Linux, ale bootloader ten może zostać zastąpiony przez dowolny bootloader zgodny ze specyfikacją multiboot. W celu spełnienia wymogów multiboot w pliku wykonywalnym z kodem systemu został umieszczony nagłówek o strukturze przedstawionej w Tabeli 3.

Tabela 3: Znaczenie poszczególnych bajtów w nagłówku w specyfikacji multiboot. [?]

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	flags	required
8	u32	checksum	required
12	u32	header_addr	if flags[16] is set
16	u32	load_addr	if flags[16] is set
20	u32	load_end_addr	if flags[16] is set
24	u32	bss_end_addr	if flags[16] is set
28	u32	entry_addr	if flags[16] is set
32	u32	mode_type	if flags[2] is set
36	u32	width	if flags[2] is set
40	u32	height	if flags[2] is set
44	u32	depth	if flags[2] is set

Pola wykorzystane w systemie to:

- magic – stała wartość określona dla specyfikacji multiboot 0x1BAADB002,
- flags – ustawione bity 0, 1, 2 odpowiadające kolejno za: ładowanie jądra systemu do adresu pamięci wyrównanego do 4KB, przekazanie do struktury multiboot informacji o dostępnej pamięci, przekazanie informacji o trybie wideo,
- checksum – wartość pola magic minus wartość pola flags,
- mode_type – wartość 0, odpowiadająca linowemu trybowi graficznemu,
- width – szerokość ekranu w pixelach ustawiona na 1024,
- height – wysokość ekranu w pixelach ustawiona na 768,

- depth – ilość bitów danych przypadających na jeden pixel ustawiona na 24.

Pozostałe pola otrzymały wartość 0.

1.2.2. Format i ładowanie plików wykonywalnych

W ramach systemu operacyjnego zdecydowano się na stworzenie wsparcia dla ładowania do pamięci systemu operacyjnego oraz wykonywania plików w formacie ELF (Executable Link Format). Format wybrano ze względu na jego powszechność w typowych uniksowych systemach operacyjnych oraz klarowność struktury (4 strukturalne części: nagłówek programu, lista jego segmentów, lista nagłówków sekcji oraz część za danymi w postaci nagłówków i sekcji⁵). Aby zapewnić poprawność działania plików w formacie ELF przeanalizowano trzy typowe sposoby linkowania wykorzystywanych przy ładowaniu plików w tym formacie (wykorzystanie adresowania bezwzględnego - hard-link, adresowanie względne PIC - Position Independent Code lub dynamiczne współdzielenie obiektów - DSO [?]). Spośród tych trzech możliwości zdecydowano się na użycie tej pierwszej, która wykorzystuje zaimplementowane wcześniej wsparcie dla pamięci wirtualnej (rozwiązanie takie jest najpopularniejsze w systemach typu UNIX posiadających takie wsparcie).

Przy ładowaniu pliku wykonywalnego w formacie ELF odczytywany jest nagłówek pliku, który sprawdzany jest pod kątem poprawności (powinien rozpoczynać się od bajtów: 0x7f, 0x45, 0x4c, 0x46). Następnie odczytywane są nagłówki programu, znajdujące się pod adresem (względem początku pliku) wskazanym w nagłówku pliku w formacie ELF, podobnie przechowywany jest tam rozmiar nagłówków oraz pozycja kończąca sekcję nagłówków. Szczegóły formatu ELF zaprezentowano w tabelach: dla nagłówka pliku - tabela 4, dla nagłówka programu - tabela 5.

Tabela 4: Format nagłówka pliku wykonywalnego ELF w 64-bitowym systemie operacyjnym. [?] [?]

Element nagłówka	Położenie pierwszego bajtu (w bajtach liczonych od 0)	Rozmiar	Znaczenie
e_ident	0	16	4 pierwsze bajty znacznik formatu, kolejne bajty: zakodowany rodzaj systemu (1 - 32 bitowy, 2 - 64 bitowy system), sposób kodowania (1 - little endian, 2 - big endian), wersja nagłówka pliku, oznaczenie rodzaju systemu operacyjnego, ostatnie 8 bajtów stanowi bufor
e_type	16	2	typ pliku (1 - realokowalny, 2 y wykonywalny, 3 - współdzielony, 4 - core)
e_machine	18	2	typ (rodzaj zestawu komend) maszyny
e_version	20	4	wersja formatu ELF
e_entry	24	8	pozycja początku programu
e_phoff	32	8	pozycja tabeli nagłówków programu
e_shoff	40	8	pozycja tabeli nagłówków sekcji
e_flags	48	4	flagi
e_ehsize	52	2	rozmiar nagłówka
e_phentsize	54	2	rozmiar nagłówków programu
e_phnum	56	2	liczba nagłówków programu
e_shentsize	58	2	rozmiar nagłówków sekcji
e_shnum	60	2	liczba nagłówków sekcji
e_shstrndx	62	2	numer sekcji w tabeli nagłówków sekcji z ich nazwami

Tabela 5: Format nagłówka programu w formacie ELF w 64-bitowym systemie operacyjnym. [?] [?]

Element nagłówka	Położenie pierwszego bajtu (w bajtach liczonych od 0)	Rozmiar	Znaczenie
p_type	0	4	typ segmentu: 0 - ignorowany, 1 - ładowany, 2 - dynamiczny, 3 - wskazanie na interpreter, 4 - komentarz
p_flags	4	4	flagi
p_offset	8	8	rozmiar offsetu dla pliku z danymi
p_vaddr	16	8	wskazanie na początek danego segmentu w przestrzeni wirtualnej
p_paddr	24	8	służy do wyznaczenia adresów bazowych
p_filesz	32	8	rozmiar segmentu w pliku
p_memsz	40	8	rozmiar segmentu w pamięci
p_align;	48	2	wymagane wyrównanie

Następnie wyznaczany jest tak zwany adres bazowy (zdefiniowany jako minimalna wartość pola *p_paddr* z nagłówków programu), za pomocą tej wartości wyznaczony zostaje rozmiar pamięci potrzebny na zapisanie całego programu (jest to największą wartość sumy *p_paddr*, *p_memsz* pomniejszonej o wartość bazową po wszystkich nagłówkach) - równanie 1 (oznaczenia zgodne z tabelami 4 oraz 5) i oznacza numer nagłówka). W kolejnym następuje zapisanie nagłówków programu w pamięci programu ładującego (jako element tablicy) oraz alokujemy pamięć potrzebną na obraz programu.

$$rozmiar_pamieci = \max_{i=1:e_phnum} (p_paddr_i + p_memsz_i - adres_bazowy) \quad (1)$$

Następnym etapem jest załadowanie segmentów programu w odpowiednie miejsce przeznaczone na cały program. Dla bezwzględnej adresacji wykorzystujemy *p_paddr* z każdego nagłówka odpowiadającego danemu segmentowi. W kolejnym etapie zerujemy obszar danych potrzebny na blok danych, a następnym kroku wypełniamy, go danymi znajdującymi się we fragmencie znajdującym się w pamięci pod adresem *p_offset*, długość tego bloku to *p_filesz*. Ostatnim etapem przygotowania pliku w formacie ELF do uruchomienia tworzymy stos dla uruchamianego programu i za jego pomocą przekazujemy programowi deskryptory standardowego wejścia, wyjścia i stderr, zestaw argumentów wywoławczych programu oraz zmienne środowiskowe [?]. Następnie aby uruchomić program następuje przeskok do początku programu wyznaczonego przez *e_entry*.

1.2.3. Interfejs graficzny

Przejsie w tryb graficzny umożliwia tworzenie interfejsów użytkownika zawierających dziesiątki elementów. Aby uniknąć manualnego ustalania pozycji każdego elementu graficznego na ekranie oraz kolejności ich wyświetlania, w stworzonym systemie zaimplementowana jest hierarchiczna struktura elementów graficznych. Zastosowane rozwiązanie wzorowane jest na przeglądarce oraz elementach HTML. Umożliwia ono dla przykładu pozycjonowanie widgetów za pomocą atrybutów *relative*, *absolute*, *static*, *fixed*, działających analogicznie jak w przeglądarce. Dzięki użyciu drzewiastej struktury elementów, wyznaczona jest także kolejność malowania na ekranie - dzieci zawsze renderowane są po rodzicu, w przypadku rodzeństwa decyduje kolejność dodania.

1.3. Istotne mechanizmy i zastosowane algorytmy

1.3.1. System plików

System plików jest kluczowym elementem istniejących systemów operacyjnych odpowiedzialnym za organizację pamięci na dysku i zapewnienie podziału zapisanych informacji na pliki i katalogi. Standardowo zapewniona jest drzewiasta struktura katalogów, a poszczególne pliki mogą być tworzone, modyfikowane, odczytywane, itd. Stworzony w ramach systemu operacyjnego system plików ma na celu połączenie prostego interfejsu BMFS [?] oraz jego lekkością i prostotą zastosowanych metod porządkowania pamięci z funkcjonalnościami dostępnymi w systemach plików dostępnych na wiodących systemach operacyjnych (drzewiasta struktura katalogów, wielkość pliku ograniczona przez rozmiar wolnej przestrzeni na dysku, a nie wartość podaną przy jego tworzeniu).

W stworzonym systemie plików położenie każdego pliku w pamięci reprezentowane jest przez strukturę *FileEntry*, zawierającą nazwę pliku, datę jego modyfikacji oraz informacje o

położeniu i rozmiarze pliku w pamięci komputera: liczbę całkowitą (typu `uint32_t`) reprezentującą początek pliku w systemie operacyjnym, liczbę całkowitą (typu `uint32_t`) reprezentującą liczbę bloków w pamięci komputera zarezerwowanych na dany plik, liczbę całkowitą (typu `uint32_t`) reprezentującą liczbę wolnych bloków przeznaczonych na dany plik oraz liczbę (typu `uint64_t`) równą wielkości pliku wyrażoną w bajtach. Bloki (każdy o rozmiarze 2KB), w których zapisany jest dany plik sąsiadują ze sobą w pamięci komputera. W pamięci zarezerwowane są również dwa dodatkowe fragmenty, które reprezentują informacje o dysku i rozmieszczenie plików na dysku. Na ogólne informacje o dysku i systemie plików zarezerwowany jest 1KB pamięci (oznaczenie wersji systemu plików oraz informacja o aktualnym katalogu, w którym pracuje system plików). Na przechowywanie informacji o plikach w postaci `FileEntry` przeznaczonych jest 311294B, co pozwala na zapisanie dokładnie 1024 plików, na jednym dysku (rozmiar `FileEntry` to 304B). W celu rozpoznania końca tablicy `FileEntry` dla danego dysku pierwszy bajt ostatniego entry jest ustawiony jako 0x00 (w systemie szesnastkowym). Puste `FileEntry` oznaczone poprzez 0x01. Oznaczenia te pozwalają usprawnić przeszukiwanie dysku.

Zdecydowano się na zachowanie interfejsu systemu plików znanego z BMFS, zatem podstawowe funkcjonalności realizowane są poprzez 8 poleceń przedstawionych w Tabeli 6 oraz dodatkowe polecenie odpowiedzialne za przechodzenie pomiędzy katalogami.

Tabela 6: Lista komend w systemie plików.

Komenda	Lista argumentów	Opis funkcjonalności
create	diskname - nazwa dysku, filename - nazwa tworzonego pliku, filesize - rozmiar pliku	Tworzenie nowego pliku oraz rezerwacja przestrzeni dla niego.
delete	diskname - nazwa dysku, filename - nazwa tworzonego pliku	Usuwanie pliku oraz uporządkowanie FileEntry na dysku.
format	diskname - nazwa dysku	Formatowanie dysku dla naszego systemu plików.
goto	diskname - nazwa dysku, directory - nazwa katalogu, do którego przechodzimy	Zapisanie na dysku informacji o katalogu, który używamy.
initialize	diskname - nazwa dysku, disksize - rozmiar dysku	Odpowiednie wypełnienie pamięci przeznaczonej na dany dysk (alokowanie pamięci na informacje o dysku i położenie plików na dysku oraz zapisanie informacji na dysku)
list	diskname - nazwa dysku	Listowanie zawartości dysku
read	diskname - nazwa dysku, filename - nazwa odczytywanego pliku	Odczytanie danych z dysku do wskazanego pliku tymczasowego
write	diskname - nazwa dysku, filename - nazwa pliku, do którego następuje zapis danych	Zapisanie danych z pliku tymczasowego do danego pliku na dysku, jeśli ilość danych do zapisania jest większa niż przestrzeń zarezerwowana na plik, usuwamy plik, a następnie tworzymy go z dwukrotnie większą (algorytm binary exponential backoff, stosowany w wielu dziedzinach informatyki [?]) ilością zarezerwowanej pamięci i powtarzamy operację zapisu, aż do skutku lub w sytuacji, gdy skończy się pamięć na dysku.

Implementacja systemu plików została stworzona w postaci biblioteki, w której występują dwa rodzaje funkcji (podstawowe odpowiadające funkcjonalnościom biblioteki oraz pomocnicze - Tab. 7-8).

Tabela 7: Lista komend w systemie plików cz. 1.

Nawa funkcja	Zwracany typ	Argumenty	Opis
handleArguments-ListQuery	void		Funkcja do zwrócenia informacji o poprawnej strukturze komendy na wypadek wywołania programu z błędnymi argumentami.
equal	int	const char* word1, const char* word2 - porównywane słowa	Funkcja to porównywania stringów zwracająca 0 - różne, 1 - równe.
isAlpha	int	char c - sprawdzana litera	Funkcja sprawdza czy znak jest literą (niezależnie od wielkości).
parseSize	unsigned long long	char* size - ciąg znaków będący argumentem wejściowym	Funkcja służąca do odczytania rozmiaru tworzonego pliku na podstawie argumentu wywołania włączając w to parsowanie jednostki.
compareEntries	static int	const void *a, const void *b - porównywane elementy w pamięci (FileEntry)	Funkcja służąca do porównywania wejść z informacji o plikach. Wykorzystywana jest do sortowania entries w momencie aktualizacji informacji o plikach znajdujących się na dysku.
toString	void	uint32_t second, uint32_t minute, uint32_t hour, uint32_t day, uint32_t month, uint32_t year - elementy składowe daty, char retString[20] - ciąg znaków, do którego będzie zapisywana data.	Funkcja służąca do zamiany daty na ciąg znaków.
handleCommand	void	int argc - liczba argumentów, char** argv - lista argumentów wywołania programu	Funkcja rozpoznaje rodzaj polecenia i zamienia go zmienną typu int, która jest wykorzystywana w dalszej części programu. Jeśli polecenie nie zostało rozpoznane to typ oznaczany jest jako 0. Powoduje to później nie wykonanie żadnej czynności i zwrócenie informacji o nierozpoznaniu polecenia użytkownikowi.

Tabela 8: Lista komend w systemie plików cz. 2.

makeItGlobal	void	int argc - liczba argumentów, char** argv - lista argumentów wywołania programu	Funkcja przepisuje argumenty wywołania programu na zmienne globalne (command, diskname, argument0-3).
closeInitializing	int	FILE *file1, FILE *file2, FILE *file3 - pliki do zamknięcia, char* buffer - bufor do zwolnienia	Funkcja odpowiedzialna za zamknięcie wszystkich otwartych plików, zwolnienie za-alokowanej pamięci na bufor danych.
closePointer	int	FILE* filePointer - wskazania na plik do zamknięcia	Funkcja do bezpiecznego zamykania pliku.
freeBuffer	int	char* buffer - bufor do zwolnienia	Funkcja do bezpiecznego zwalniania buforu.
openGeneralFile	int	char* argument0 - nazwa pliku, char* message - element składowy informacji do przekazania na IO, FILE *filePointer - wskazanie na plik do otwarcia, int retval - wartość wskazująca stan programu 0 - brak błędu.	Funkcja otwierająca plik do zapisania na dysku na trwałe. Takimi plikami mogą być pliki Boot Recorder'a, kernal'a itp.
open	int		Funkcja odpowiadająca za otwarcie systemu plików.
find	int	char* argument0 - nazwa pliku do znalezienia, FileEntry* fileEntry - wskazanie na fragment pamięci, w którym będziemy przechowywać informacje o znalezionym pliku.	Funkcja służąca do znalezienia pozycji entry z informacjami o lokalizacji pliku w pamięci.
close	int		Zamykanie dostępu do dysku.

1.3.2. Zarządzanie procesami

W ramach zarządzania procesami zdecydowano się na możliwie duże uproszczenie oprogramowania związanego z zarządzaniem procesami (ze względu na założenie o lekkości systemu operacyjnego). Z tego powodu spośród dwóch standardowych podejść do zarządzania procesami (Cooperative Multitasking i Preemptive Multitasking) wybrano uproszczoną wersję Cooperative Multitasking. Podejście to umożliwia ona zachowanie lekkości pamięciowej systemu operacyjnego częściowo kosztem wydajności. Realizacja takiego mechanizmu opiera się na wykonywaniu procesów aż do momentu zwolnieniu zasobów przez wykonujący się proces. Taka

sytuacja następuje, gdy program, kończy swoje działania lub wstrzymuje swoje wykonanie poprzez polecenie *yield*.

Zapewnienie funkcjonowania takiego mechanizmu można uzyskać poprzez stworzenie struktur odpowiadających za przechowywanie odpowiednich rejestrów procesora definiujących dany proces oraz struktury przechowujących dane procesów w postaci listy wskaźnikowej. Dla ułatwienia działania systemu wykorzystano listę dwukierunkową (łatwiejszy proces usuwania wpisów z listy). Dodatkowo zaimplementowano funkcje służące do inicjalizowania zarządzania procesami (*initializeTasking*), tworzenia pojedynczego zadania (*createTask*) oraz funkcję dokonującą wstrzymania wykonywania procesu (*yield*) wraz z funkcją służącą do zmiany wykonywanego procesu (*switchTask*).

Inicjalizacja zarządzania procesami polega na wypełnieniu danymi odpowiednich rejestrów, utworzeniu listy procesów zawierających dwa procesy, stanowiące ogon i głowę listy. Procesy powinny wskazywać na siebie na wzajem, jako na kolejny proces do wykonania. Funkcja *createTask* powinna tworzyć proces, wypełniając wartości rejestrów odpowiednimi wartościami, wskazywać miejsce w pamięci z zadaniem do wykonania oraz alokować potrzebną pamięć. Po wykonaniu tych zadań następuje wstawienie nowo utworzonego procesu do listy zarejestrowanych procesów. Funkcja *yield* przesunięcia wskaźnika na liście zadań do wykonania na kolejną pozycję oraz deleguje zadanie do funkcji *switchTask*, która dokonuje podmiany zawartości rejestru reprezentującego nowy proces, na wartości rejestrów ostatnio wykonywanego procesu.

Dodatkowo należy zapewnić mechanizm pozwalający na zakończenie wykonywania aktualnie działającego procesu. Poprzez zwolnienie pamięci, przełączenie na kolejny wykonywany proces oraz usunięcie wskazanego procesu z listy procesów, do wykonania.

2. Funkcje i struktury

2.1. headers/clock_driver.h

2.1.1. get_clock_ticks

Typ: uint64_t

Opis:

Funkcja służąca do pobrania liczby cykli zegara, które minęły od pierwszego uruchomienia systemu.

2.1.2. handle_clock_interrupt

Typ: void

Opis:

Funkcja służąca do obsługi przerwania dla zegara systemowego. Liczba cykli zegara inkrementowana jest o jeden.

2.2. headers/deque.h

Moduł zawierający własną implementację struktury *Deque* - kolejka w języku C. Stanowi element pomocniczy dla implementacji niektórych mechanizmów systemu operacyjnego.

2.2.1. deque_create

Typ: Deque*

Opis:

Funkcja tworząca i inicjalizująca strukturę *Dequeue*.

Argumenty:

- element_size (typ: size_t) - rozmiar kolejki.

2.2.2. deque_push_left

Typ: void

Opis:

Funkcja służy do dodania elementu po lewej stronie kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, do której przekazujemy element,
- element (typ: void*) - wskaźnik na przekazywany element.

2.2.3. deque_push_right

Typ: void

Opis:

Funkcja służy do dodania elementu po prawej stronie kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, do której przekazujemy element,
- element (typ: void*) - wskaźnik na przekazywany element.

2.2.4. deque_peek_left

Typ: void*

Opis:

Funkcja służy do odczytania elementu po lewej stronie kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, z której odczytywany jest element.

2.2.5. deque_peek_right

Typ: void*

Opis:

Funkcja służy do odczytania elementu po lewej stronie kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, z której odczytywany jest element.

2.2.6. deque_pop_left

Typ: void*

Opis:

Funkcja służy do odczytania elementu z lewej strony kolejki, odczytany element jest usuwany z kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, z której usuwany jest element.

2.2.7. deque_pop_right

Typ: void*

Opis:

Funkcja służy do zdjęcia elementu z prawej strony kolejki, odczytany element jest usuwany z kolejki.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę, z której usuwany jest element.

2.2.8. deque_destory

Typ: void

Opis:

Funkcja służy do zwolnienia pamięci zaalokowanej na strukturę *Dequeue*.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę do usunięcia,

2.2.9. deque_is_empty

Typ: uint8_t

Opis:

Funkcja służy do sprawdzenia czy kolejka jest pusta.

Argumenty:

- deque (typ: Deque*) - wskaźnik na kolejkę,

2.2.10. struct Deque

Opis:

Struktura reprezentująca kolejkę. Posiada wskazanie na skrajnie lewy i skrajnie prawy element kolejki, wskaźniki na początek i koniec bufora zaalokowanego na kolejkę oraz informację o rozmiarze elementu przechowywanego w kolejce.

Pola:

- void* left
- void* right
- void* buffer
- void* buffer_end

- size_t length
- size_t element_size

2.3. headers/desktop.h

2.3.1. create_desktop

Typ: Widget*

Opis:

Funkcja zwracająca Widget reprezentujący aktualny stan (zawartość wyświetlanych pikseli) dla widoku monitora.

2.4. headers/desktop_selected_zone.h

2.4.1. create_selected_zone

Typ: Widget*

Opis:

Funkcja służąca do stworzenia struktury Widget reprezentującej wybraną część ekranu.

Argumenty:

- parent (typ: Widget*) - wskaźnik na Widget, w którym występuje kliknięcie,
- last_left_click_x (typ: uint32_t*) - współrzędna x miejsca na ekranie, w którym nastąpiło ostatnie przyciśnięcie myszy (wyrażona w pikselach),
- last_left_click_y (typ: uint32_t*) - współrzędna y miejsca na ekranie, w którym nastąpiło ostatnie przyciśnięcie myszy (wyrażona w pikselach),
- is_mouse_clicked (typ: uint8_t*) - informacja o przyciśnięciu.

2.5. headers/event.h

2.5.1. event_init

Typ: void

Opis:

Funkcja inicjalizująca strukturę reprezentującą wydarzenie, do którego doszło na ekranie.

Argumenty:

- event (typ: Event*) - wskaźnik na strukturę reprezentującą wydarzenie,
- mouse_x (typ: uint32_t) - współrzędna x myszy,
- mouse_y (typ: uint32_t) - współrzędna y myszy.

2.5.2. event_stop_propagation

Typ: void

Opis:

Funkcja odpowiedzialna za wstrzymanie propagacji zdarzenia.

Argumenty:

- event (typ: Event*) - wydarzenie do wstrzymania.

2.5.3. struct Event

Opis:

Struktura reprezentująca wydarzenie, do którego doszło na ekranie.

Pola:

- uint32_t mouse_x - współrzędna x dla wydarzenia,
- uint32_t mouse_y - współrzędna y dla wydarzenia,
- uint8_t is_bubbling - flaga reprezentująca informację o tym, czy zdarzenie ma być dalej propagowane.
- uint8_t is_captured - czy wydarzenie zostało przechwycone,
- void* target - wskaźnik na cel zdarzenia.

2.6. headers/gdt.h

2.6.1. fill_gdt

Typ: void

Opis:

Funkcja dokonująca wypełnienia struktury GDT.

Argumenty:

- gdt (typ: Memory_Segment*) - segment pamięci do wypełnienia.

2.6.2. load_gdt

Typ: void

Opis:

Funkcja służąca do załadowania struktury GDT.

2.6.3. struct Memory_Segment

Struktura reprezentująca segment pamięci.

Pola:

- uint16_t limit_low
- uint16_t base_low

- uint8_t base_middle
- uint8_t access
- uint8_t granularity
- uint8_t base_high

2.6.4. struct GDT_Register

Struktura reprezentująca rejestr GDT pamięci.

Pola:

- uint16_t gdt_size - rozmiar rejestru,
- uint32_t gdt_pointer - wskaźnik na segment pamięci.

2.7. headers/graphics_mode.h

2.7.1. set_graphics_mode

Typ: void

Opis:

Funkcja służąca do przejścia w tryb graficzny.

Argumenty:

- boot_info (typ: void*).

2.7.2. add_drag_observer

Typ: void

Opis:

Dodanie w trybie graficznym obserwatora zdarzeń związanych z przeciąganiem elementów do wskazanego Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na strukturę Widget, do której dodajemy obserwator.

2.7.3. add_release_observer

Typ: void

Opis:

Dodanie w trybie graficznym obserwatora zdarzeń związanych ze zwolnieniem przyciska myszy do wskazanego Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na strukturę Widget, do której dodajemy obserwator.

2.7.4. remove_drag_observer

Typ: void

Opis:

Usunięcie w trybie graficznym obserwatora zdarzeń związanych z przeciąganiem elementów do wskazanego Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na strukturę Widget, z której usuwamy obserwator.

2.7.5. remove_release_observer

Typ: void

Opis:

Usunięcie w trybie graficznym obserwatora zdarzeń związanych z przeciąganiem elementów do wskazanego Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na strukturę Widget, z której usuwamy obserwator.

2.7.6. render

Typ: void

Opis:

Funkcja renderująca stan monitora.

2.7.7. handle_mouse_event

Typ: void

Opis:

Obsługa zdarzeń związanych z działaniem myszy.

Argumenty:

- mouse_event (typ: MouseEvent*) - wskaźnik na zdarzenie,

2.7.8. get_mouse_x

Typ: uint32_t

Opis:

Funkcja służąca do pobrania x-owej współrzędnej położenia myszy wyrażona w pikselach.

2.7.9. get_mouse_y

Typ: uint32_t

Opis:

Funkcja służąca do pobrania y-owej współrzędnej położenia myszy wyrażona w pikselach

2.8. headers/icons/cursor.h

Plik przechowuje informacje o wyglądzie kursora myszy - obraz w formacie BGR zapisany w pliku nagłówkowym.

2.9. headers/icons

Katalog zawiera pliki nagłówkowe reprezentujące poszczególne elementy graficzne wykorzystane w interfejsie użytkownika (graficznym). Poszczególne pliki zawierają tablice, w których zapisane są dane obrazów w formacie BGR.

2.10. headers/icons.h

2.10.1. render_icons

Typ: void

Opis:

Funkcja służąca renderowania wyglądu ikony w trybie graficznym systemu operacyjnego.

Argumenty:

- `backbuffer` (typ: `uint8_t*`) - bufor reprezentujący zawartość ikony,
- `is_dragging_icons` (typ: `uint8_t`) - flaga reprezentująca informację czy ikona jest przesuwana,
- `drag_x` (typ: `int32_t`) - współrzędna x zdarzenia związanego z przesuwaniem,
- `drag_y` (typ: `int32_t`) - współrzędna y zdarzenia związanego z przesuwaniem.

2.10.2. select_icons

Typ: void

Opis:

Funkcja odpowiadająca za reprezentację zdarzenia wyboru ikony.

Argumenty:

- `x0` (typ: `uint32_t`),
- `y0` (typ: `uint32_t`),
- `x1` (typ: `uint32_t`),
- `y1` (typ: `uint32_t`).

2.10.3. create_icons

Typ: void

Opis:

Funkcja odpowiedzialna za stworzenie Widgetu dla ikony. **Argumenty:**

- `parent` (typ: `Widget*`) - wskaźnik na Widget.

2.10.4. stop_dragging_icons

Typ: void

Opis:

Funkcja odpowiedzialna za wstrzymanie przesuwania ikon.

2.11. headers/image.h

2.11.1. image_create

Typ: void

Opis:

Funkcja odpowiedzialna za tworzenie Widgetu reprezentującego obraz na ekranie.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- width (typ: uint32_t) - szerokość obrazu,
- height (typ: uint32_t) - wysokość obrazu,
- bitmap (typ: uint8_t*) - tablica reprezentująca obraz,
- bitmask (typ: uint8_t*) - maska bitowa reprezentująca kształt obrazu.

2.11.2. image_destory

Typ: void

Opis:

Usuwanie obrazu z ekranu w trybie graficznym.

2.12. headers/interrupts.h

2.12.1. fill_idt

Typ: void

Opis:

Funkcja służąca do wypełnienia struktury reprezentującej przerwanie systemowe w trybie 64-bitowym.

Argumenty:

- idt (typ: InterruptDescriptor64*) - wskaźnik na strukturę reprezentującą przerwanie systemowe w trybie 64-bitowym,
- code_segment (typ: uint16_t) - wskaźnik na segment kodu, z którego przyszło przerwanie.

2.12.2. activate_idt

Typ: void

Opis:

Funkcja służąca do aktywacji przerwania systemowego.

2.12.3. struct Gate_Descriptor

Pola:

- uint16_t handler_low
- uint16_t code_segment
- uint8_t reserved
- uint8_t access
- uint16_t handler_high

2.12.4. struct IDT_Pointer

Struktura reprezentująca wskaźnik na przerwanie systemowe.

Pola:

- uint16_t size - rozmiar,
- uint64_t base.

2.12.5. struct InterruptDescriptor64

Struktura reprezentująca opis przerwania systemowego w trybie 64-bitowym.

Pola:

- uint16_t offset_1 - offset bits 0..15,
- uint16_t selector - a code segment selector in GDT or LDT,
- uint8_t ist - bits 0..2 holds Interrupt Stack Table offset, rest of bits zero.,
- uint8_t type_attributes - gate type, dpl, and p fields,
- uint16_t offset_2 - offset bits 16..31,
- uint32_t offset_3 - offset bits 32..63,
- uint32_t zero - reserved,

2.13. headers/interrupt_handlers.h

Funkcje służące do obsługi przerw systemowych oznaczanych przez bajty 0x00-0x6F. Wszystkie funkcje mają typ void i nie przyjmują argumentów wywołania. Przykład interfejsu funkcji zaprezentowano poniżej (przez nn należy rozumieć liczbę w zapisie szesnastkowym reprezentującą numer przerwania):

2.13.1. handle_exception0xnn

Typ: void

2.13.2. ignore_interrupt

Typ: void

Opis:

Funkcja odpowiedzialna za ignorowanie przerwania systemowego.

2.14. headers/keyboard_driver.h

2.14.1. handle_keyboard_interrupt

Typ: void

Opis:

Funkcja odpowiedzialna za obsługę wydarzeń związanych ze sterownikiem klawiatury (przyścisnięcie klawisza na klawiaturze).

2.14.2. activate_keyboard

Typ: void

Opis:

Funkcja odpowiedzialna za aktywację klawiatury (aktywacja przerwania związanych z klawiaturą, odczytanie i ustawienie bajtu reprezentującego komendę).

2.15. headers/line_A20.h

2.15.1. check_A20

Typ: void

Opis:

Funkcja odpowiedzialna za sprawdzenie linii A20.

2.16. headers/list.h

2.16.1. list_create

Typ: List*

Opis:

Funkcja odpowiedzialna za stworzenie struktury listy, służącej jako element pomocniczy do budowy systemu operacyjnego.

2.16.2. list_append

Typ: void

Opis:

Funkcja odpowiedzialna za dodanie elementu do listy.

Argumenty:

- list (typ: List*) - wskaźnik na listę,
- element (typ: void*) - dodawany element.

2.16.3. list_pop

Typ: void

Opis:

Funkcja służąca do odczytania wskazanego elementu listy (i jego usunięcia).

Argumenty:

- list (typ: List*) - wskaźnik na listę,
- index (typ: int32_t) - odczytywany indeks z listy.

2.16.4. list_insert

Typ: void

Opis:

Argumenty: Funkcja odpowiedzialna za wstawienie elementu do listy na określoną pozycję.

- list (typ: List*) - wskaźnik na listę,
- index (typ: size_t) - indeks w liście,
- element (typ: void*) - wstawiany element.

2.16.5. list_get

Typ: void*

Opis:

Funkcja służąca do odczytania wskazanego elementu listy.

Argumenty:

- list (typ: List*) - wskaźnik na listę,
- index (typ: size_t) - indeks, z którego chcemy odczytać.

2.16.6. list_remove

Typ: void

Opis:

Funkcja służąca do usunięcia wskazanego elementu listy.

Argumenty:

- list (typ: List*) - wskaźnik na listę,
- element (typ: void*) - wskaźnik na element do usunięcia.

2.16.7. list_destroy

Typ: void

Opis:

Funkcja służąca do usunięcia listy z pamięci.

Argumenty:

- list (typ: List*) - wskaźnik na listę.

2.16.8. list_find_index

Typ: size_t

Opis:

Funkcja służąca do znalezienia indeksu, pod którym znajduje się wskazany element.

Argumenty:

- list (typ: List*) - wskaźnik na listę,
- element (typ: void*) - wskaźnik na element do znalezienia.

2.16.9. struct List

Struktura reprezentująca listę wskaźnikową. Przechowujemy w niej rozmiar listy, jej maksymalny rozmiar oraz wskaźnik na pierwszy element listy. **Pola:**

- size_t length
- size_t max_size
- void** list

2.17. headers/long_mode.h

2.17.1. detect_long_mode

Typ: void

Opis:

Funkcja służąca do wykrycia long_mode.

2.18. headers/memory.h

2.18.1. memcpy

Typ: void

Opis:

Funkcja służąca do kopiowania wybranego fragmentu pamięci (źródło) do docelowego miejsca (cel).

Argumenty:

- to (typ: void*) - wskaźnik na cel,
- from (typ: void*) - wskaźnik na źródło,
- n (typ: size_t) - rozmiar pamięci do skopiowania w bajtach.

2.18.2. `init_memory_blocks`

Typ: void

Opis:

Inicjalizacja bloków pamięci.

2.18.3. `malloc`

Typ: void*

Opis:

Funkcja służąca do alokacji fragmentu pamięci na o wskazanym rozmiarze.

Argumenty:

- size (typ: `size_t`) - rozmiar pamięci do zaalokowania.

2.18.4. `free`

Typ: void

Opis:

Funkcja służąca do zwolnienia wskazanego fragmentu pamięci.

Argumenty:

- start (typ: void*) - wskaźnik na fragment pamięci do zwolnienia.

2.18.5. `bitmask_memcpy`

Typ: void*

Opis:

Argumenty: Funkcja służąca do skopiowania fragmentów pamięci zdefiniowanych przez maskę bitową.

- to (typ: void*) - cel,
- from (typ: void*) - źródło,
- bitmask (typ: void*) - maska bitowa,
- n (typ: `size_t`) - rozmiar maski bitowej (w bajtach),

2.19. `headers/mouse_driver.h`

2.19.1. `activate_mouse`

Typ: void

Opis:

Funkcja służąca do aktywacji sterownika myszy (inicjalizacja kolejki z wydarzeniami).

Argumenty:

- queue (typ: `SynchronizedQueue*`) - kolejka wydarzeń.

2.19.2. `handle_mouse_interrupt`

Typ: void

Opis:

Funkcja służąca do obsłużenia przerwania związanego z wydarzeniem wywołanym przez wykorzystanie myszy.

2.19.3. `enum MouseEventType`

Enumerator typów wydarzeń związanych z przyciśnięciem myszy.

Pola:

- `MouseDown` - przeciąganie,
- `MouseLeftClick` - przyciśnięcie lewego przycisku myszy,
- `MouseRightClick` - przyciśnięcie prawego przycisku myszy,
- `MouseLeftRelease` - zwolnienie lewego przycisku myszy,
- `MouseRightRelease` - zwolnienie prawego przycisku myszy.

2.19.4. `struct MouseEvent`

Struktura reprezentująca zdarzeń związanych z przyciśnięciem jednego z przycisków myszy.

Pola:

- `MouseEventType type` - typ wydarzenia,
- `int32_t x` - współrzędna x zdarzenia,
- `int32_t y` - współrzędna y zdarzenia.

2.20. `headers/paging.h`

2.20.1. `setup_paging`

Typ: void

Opis:

Funkcja służąca do ustawienia stronicowania pamięci.

2.21. `headers/port.h`

2.21.1. `port_write8`

Typ: void

Opis:

Funkcja odpowiedzialna za zapisanie 8 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint8_t) - dane do zapisania.

2.21.2. port_write16

Typ: void

Opis:

Funkcja odpowiedzialna za zapisanie 16 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint16_t) - dane do zapisania.

2.21.3. port_write32

Typ: void

Opis:

Funkcja odpowiedzialna za zapisanie 32 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint32_t) - dane do zapisania.

2.21.4. port_slow_write8

Typ: void

Opis:

Funkcja odpowiedzialna za wolne zapisanie (skok do portu i powrót) 8 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint8_t) - dane do zapisania.

2.21.5. port_slow_write16

Typ: void

Opis:

Funkcja odpowiedzialna za wolne zapisanie (skok do portu i powrót) 16 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint16_t) - dane do zapisania.

2.21.6. port_slow_write32

Typ: void

Opis:

Funkcja odpowiedzialna za wolne (skok do portu i powrót) zapisanie 32 bajtów danych na wskazany port.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port,
- data (typ: uint32_t) - dane do zapisania.

2.21.7. port_read8

Typ: uint8_t

Opis:

Funkcja odpowiedzialna za odczytanie 8 bajtów danych ze wskazanego portu.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port.

2.21.8. port_read16

Typ: uint16_t

Opis:

Funkcja odpowiedzialna za odczytanie 16 bajtów danych ze wskazanego portu.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port.

2.21.9. port_read32

Typ: uint32_t

Opis:

Funkcja odpowiedzialna za odczytanie 32 bajtów danych ze wskazanego portu.

Argumenty:

- port (typ: uint16_t) - liczba reprezentująca port.

2.22. headers/synchronized_queue.h

2.22.1. synchronized_queue_create

Typ: SynchronizedQueue*

Opis:

Funkcja służąca do stworzenia instancji zsynchronizowanej kolejki.

Argumenty:

- element_size (typ: size_t) - rozmiar elementów przeznaczonych do wstawiania do kolejki.

2.22.2. `synchronized_queue_put`

Typ: void

Opis:

Funkcja służąca do wstawienia elementu do wskazanej zsynchronizowanej kolejki.

Argumenty:

- `queue` (typ: `SynchronizedQueue*`) - wskaźnik na kolejkę, do której wstawiamy element,
- `element` (typ: `void*`) - wskaźnik na wstawiany element.

2.22.3. `synchronized_queue_get`

Typ: void*

Opis:

Funkcja służąca do pobrania elementu ze wskazanej zsynchronizowanej kolejki.

Argumenty:

- `queue` (typ: `SynchronizedQueue*`) - wskaźnik na kolejkę, z której pobieramy element.

2.22.4. `synchronized_queue_is_empty`

Typ: uint8_t

Opis:

Funkcja służąca do sprawdzenia czy zsynchronizowana kolejka jest pusta.

Argumenty:

- `queue` (typ: `SynchronizedQueue*`) - wskaźnik na kolejkę, która jest sprawdzana.

2.22.5. `synchronized_queue_peek_last`

Typ: void*

Opis:

Funkcja służąca do pobrania ostatniego elementu ze wskazanej zsynchronizowanej kolejki.

Argumenty:

- `queue` (typ: `SynchronizedQueue*`) - wskaźnik na kolejkę, z której pobieramy element.

2.22.6. `synchronized_queue_destory`

Typ: void

Opis:

Funkcja służąca do usuwania przekazanej jako argument kolejki zsynchronizowanej poprzez zwolnienie pamięci zaalokowanej na elementy kolejki.

Argumenty:

- `queue` (typ: `SynchronizedQueue*`) - wskaźnik na usuwaną kolejkę zsynchronizowaną.

2.22.7. struct SynchronizedQueue

Struktura reprezentująca zsynchronizowaną kolejkę.

Pola:

- Deque* first_deque - wskaźnik na pierwszą kolejkę,
- Deque* second_deque - wskaźnik na drugą kolejkę,
- uint8_t is_using_first_deque - flaga reprezentująca fakt czy używana jest pierwsza kolejka,
- uint8_t should_reset - flaga reprezentująca, czy kolejka powinna być zresetowana.

2.23. headers/tss.h

2.23.1. fill_tss

Typ: void

Opis:

Funkcja służąca do wypełniania struktury reprezentującej Task State Statement.

2.23.2. struct tss

Struktury reprezentującej Task State Statement, zawierająca informacje potrzebne do przywrócenia zadania po jego przerwaniu.

Pola:

- uint32_t reserved
- uint64_t rsp0
- uint64_t rsp1
- uint64_t rsp2
- uint64_t reserved2
- uint64_t ist1
- uint64_t ist2
- uint64_t ist3
- uint64_t ist4
- uint64_t ist5
- uint64_t ist6
- uint64_t ist7
- uint64_t reserved3
- uint16_t reserved4
- uint16_t io_map_base_address

2.24. headers/types.h

Plik zawiera definicje makr i definicji typów wykorzystywanych do ułatwienia pracy na systemem operacyjnym.

2.25. headers/utils.h

2.25.1. printf

Typ: void

Opis:

Własna implementacja funkcji printf stworzona do uproszczenia debuggowania w trakcie tworzenia systemu operacyjnego (poprzez zapisanie w określonym miejscu pamięci).

Argumenty:

- str (typ: char*) - wskaźnik na informację do zapamiętania.

2.25.2. printfHex

Typ: void

Opis:

Funkcja pomocnicza służąca do zapisania w pamięci liczby w systemie szesnastkowym (w postaci napisu).

Argumenty:

- key (typ: uint8_t) - liczba do zapisania w pamięci.

2.25.3. printfHex16

Typ: void

Opis:

Funkcja pomocnicza służąca do zapisania w pamięci liczby (zapisanej na 2 bajtach w pamięci) w systemie szesnastkowym (w postaci napisu).

Argumenty:

- key (typ: uint16_t) - liczba do zapisania w pamięci.

2.25.4. printfHex32

Typ: void

Opis:

Funkcja pomocnicza służąca do zapisania w pamięci liczby (zapisanej na 4 bajtach w pamięci) w systemie szesnastkowym (w postaci napisu).

Argumenty:

- key (typ: uint32_t) - liczba do zapisania w pamięci.

2.25.5. printfHex64

Typ: void

Opis:

Funkcja pomocnicza służąca do zapisania w pamięci liczby (zapisanej na 8 bajtach w pamięci) w systemie szesnastkowym (w postaci napisu).

Argumenty:

- key (typ: uint64_t) - liczba do zapisania w pamięci.

2.25.6. printf_memory

Typ: void

Opis:

Argumenty: Własna implementacja funkcji printf stworzona do uproszczenia debuggowania w trakcie tworzenia systemu operacyjnego (poprzez zapisanie w określonym miejscu pamięci).

- start (typ: uint64_t) - wskaźnik na początek pamięci do zapisania,
- end (typ: uint64_t) - wskaźnik na koniec pamięci do zapisania.

2.25.7. clearScreen

Typ: void

Opis:

Funkcja służąca do wyczyszczenia ekranu debuggowania. Wyświetlenia na nim pustych linii.

2.25.8. min

Typ: int32_t

Opis:

Implementacja funkcji minimum.

Argumenty:

- x (typ: int32_t),
- y (typ: int32_t).

2.25.9. max

Typ: int32_t

Opis:

Implementacja funkcji maksimum.

Argumenty:

- x (typ: int32_t),
- y (typ: int32_t).

2.25.10. round

Typ: int32_t

Opis:

Implementacja zaokrąglenia liczby zmiennoprzecinkowej.

Argumenty:

- x (typ: double).

2.25.11. in_range

Typ: int32_t

Opis:

Funkcja sprawdzająca czy liczba znajduje się w zadanym jako argument zakresie.

Argumenty:

- min_val (typ: int32_t) - liczba reprezentująca dolną granicę zakresu,
- max_val (typ: int32_t) - liczba reprezentująca górną granicę zakresu,
- val (typ: int32_t) - wartość sprawdzana czy jest w zakresie.

2.26. headers/widget.h

2.26.1. init_widget

Typ: void

Opis:

Funkcja służąca do zainicjalizowania Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.

2.26.2. widget_destroy

Typ: void

Opis:

Funkcja służąca do usunięcia z pamięci struktury reprezentującej Widget.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget do usunięcia.

2.26.3. widget_render

Typ: void

Opis:

Funkcja służąca do renderowania Widget'u w oparciu o bufor przekazany jako argument funkcji reprezentujący aktualną zawartość Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- buffer (typ: uint8_t*) - wskaźnik na fragment pamięci reprezentujący zawartość fragmentu obrazu do wyświetlenia na monitorze.

2.26.4. widget_on_mouse_down

Typ: void

Opis:

Funkcja służąca do obsłużenia przyciśnięcia klawisza myszy w obszarze Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- event (typ: Event*) - wskaźnik na Event związany z przycisnięciem myszy.

2.26.5. widget_on_mouse_enter

Typ: void

Opis:

Funkcja służąca do najechania kursorem na obszar Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- event (typ: Event*) - wskaźnik na zdarzenie reprezentujące najechanie na obszar Widgetu.

2.26.6. widget_on_mouse_leave

Typ: void

Opis:

Funkcja służąca do obsłużenia zdarzenia opuszczenia przez kursor obszaru Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- event (typ: Event*) - wskaźnik na zdarzenie, które miało miejsce.

2.26.7. widget_on_mouse_up

Typ: void

Opis:

Funkcja służąca do obsłużenia puszczenia klawisza myszy w obszarze Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- event (typ: Event*) - wskaźnik na zdarzenie, które miało miejsce.

2.26.8. widget_set_width

Typ: void

Opis:

Funkcja służąca do ustalenia szerokości Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- width (typ: uint32_t) - szerokość.

2.26.9. widget_set_height

Typ: void

Opis:

Funkcja służąca do ustalenia wysokości Widgetu.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- height (typ: uint32_t) - docelowa wysokość Widget'u.

2.26.10. widget_set_top

Typ: void

Opis:

Funkcja służąca do ustalenia położenia górnej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- top (typ: uint32_t) - położenie górnej krawędzi Widget'u wyrażona w pikselach.

2.26.11. widget_clear_top

Typ: void

Opis:

Funkcja służąca do wyczyszczenia górnej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.

2.26.12. widget_set_bottom

Typ: void

Opis:

Funkcja służąca do ustalenia położenia dolnej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- bottom (typ: int32_t) - położenie dolnej krawędzi Widget'u wyrażona w pikselach.

2.26.13. widget_clear_bottom

Typ: void

Opis:

Funkcja służąca do wyczyszczenia dolnej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.

2.26.14. widget_set_left

Typ: void

Opis:

Funkcja służąca do ustalenia położenia lewej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget,
- left (typ: int32_t) - położenie lewej krawędzi Widget'u wyrażona w pikselach.

2.26.15. widget_clear_left

Typ: void

Opis:

Funkcja służąca do wyczyszczenia lewej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.

2.26.16. widget_set_right

Typ: void

Opis:

Funkcja służąca do ustalenia położenia prawej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.
- right (typ: int32_t) - położenie prawej krawędzi Widget'u wyrażona w pikselach.

2.26.17. widget_clear_right

Typ: void

Opis:

Funkcja służąca do wyczyszczenia lewej krawędzi Widget'u.

Argumenty:

- widget (typ: Widget*) - wskaźnik na Widget.

2.26.18. widget_append_child

Typ: void

Opis:

Funkcja służąca do wydłużenia listy potomków Widget'ów o nowy Widget w hierarchicznej (drzewiastej) strukturze Widget'ów.

Argumenty:

- parent (typ: Widget*) - wskaźnik na Widget, do którego dodawany będzie potomek,
- child (typ: Widget*) - dodawany Widget (potomek).

2.26.19. enum WidgetPositionType

Enumerator reprezentujący możliwe pozycjonowania warstwy (Widget'u) względem jego bezpośredniego potomka.

Pola:

- PositionStatic,
- PositionAbsolute,
- PositionRelative,
- PositionFixed.

2.26.20. enum WidgetSizeType

Enumerator reprezentujący sposób skalowania warstwy.

Pola:

- SizeAuto,
- SizeFitContent,
- SizeStatic.

2.26.21. struct WidgetStyle

Struktura reprezentująca styl Widget'u określający pozycję warstwy. **Pola:**

- WidgetPositionType position
- int32_t top
- int32_t bottom
- int32_t left
- int32_t right
- uint8_t position_flags - informacje o tym, która wartość top, bottom, left, right będzie wykorzystana do obliczenia położenia warstwy,
- uint32_t background_color - kolor tła.
- uint8_t* background_image - wskaźnik na obraz reprezentujący tło.
- uint8_t* background_bitmask - wskaźnik na bitmaskę dla danej warstwy.
- double background_opacity - przezroczystość tła.

2.26.22. struct Widget

Struktura reprezentująca Widget.

Pola:

- int32_t x - wyrażone w pikselach współrzędna x warstwy,
- int32_t y - wyrażone w pikselach współrzędna y warstwy,
- uint32_t width - szerokość warstwy,
- uint32_t height - wysokość warstwy,
- WidgetSizeType width_type - sposób skalowania szerokości warstwy.
- WidgetSizeType height_type - sposób skalowania wysokości warstwy.
- uint8_t is_hidden - flaga o informacji czy warstwa jest schowana.
- List* children - lista potomków warstwy.
- void* extra_data
- void* render_data
- WidgetStyle style - styl warstwy,
- struct Widget* parent
- void (*render)(struct Widget* widget, uint8_t* buffer)
- void (*on_mouse_down)(struct Widget* widget, Event* event) - funkcja obsługująca przyciśnięcie myszy,
- void (*on_mouse_enter)(struct Widget* widget, Event* event) - funkcja obsługująca najeżdżanie myszą na warstwę,
- void (*on_mouse_leave)(struct Widget* widget, Event* event) - funkcja obsługująca opuszczenie myszą warstwy,
- void (*on_mouse_up)(struct Widget* widget, Event* event) - funkcja obsługująca puszczenie klawisza myszy,
- void (*on_mouse_move)(struct Widget* widget, Event* event) - funkcja obsługująca ruch myszy,

2.27. headers/windows.h

2.27.1. init_windows

Typ: void

Opis:

Funkcja służąca do inicjalizacji okna przeznaczonego do wyświetlenia na ekranie.

Argumenty:

- desktop (typ: Widget*) - główna warstwa na ekranie,
- windows_container (typ: Widget*) - kontener na okna do wyświetlenia,
- bottom_bar (typ: Widget*) - warstwa reprezentująca dolny pasek,

2.27.2. create_window

Typ: Widget*

Opis:

Funkcja służąca do stworzenia okna.

Argumenty:

- icon (typ: uint8_t*) - ikona do wyświetlenia w postaci wskaźnika na fragment pamięci,
- icon_bitmask (typ: uint8_t*) - maska bitowa reprezentująca kształt danej warstwy.

2.28. headers/windows_widget.h

2.28.1. create_window

Typ: Widget*

Opis:

Funkcja służąca do stworzenia okna.

2.29. lib/file_system_lib.h

2.30. Funkcje

2.30.1. callLibrary

Typ: int

Opis:

Funkcja stanowiąca wywołanie biblioteki służącej do obsługi systemu plików.

Argumenty:

- argn (typ: int) - liczba argumentów wywołania,
- args[8][256] (typ: char) - lista argumentów wywołania.

2.31. lib/multiTasking.h

2.32. Funkcje

2.32.1. mainFunction

Typ: void

Opis:

Funkcja pomocnicza przy procesie debuggowania tworzonego oprogramowania.

2.32.2. newFunction

Typ: void

Opis:

Funkcja pomocnicza przy procesie debuggowania tworzonego oprogramowania.

2.32.3. createTask

Typ: void

Opis:

Funkcja odpowiadająca za uzupełnienie wartości poszczególnych pól w strukturze *Task*.

Argumenty:

- task (typ: Task *) - wskaźnik do struktury Task,
- void (*main)() - wskazanie na funkcję do wykonywania,
- flags (typ: uint32_t) flags - zawartość pola flagi w strukturze rejestrów,
- pagedir (typ: uint32_t) - wartość przypisywana do rejestru cr3,
- isMain (typ: uint32_t) - informacja o tym czy zadanie jest głównym procesem,

2.32.4. initTasking

Typ: void

Opis:

Funkcja odpowiedzialna za inicjalizację nadzoru nad wykonywanymi zadaniami (zarejestrowanie głównego procesu, przygotowanie listy na nowe procesy).

2.32.5. addTask

Typ: void

Opis:

Funkcja odpowiedzialna za dodawanie nowego zadania do listy.

2.32.6. removeTask

Typ: void

Opis:

Funkcja odpowiedzialna za dodawanie nowego zadania do listy.

2.32.7. yield

Typ: void

Opis:

Funkcja umożliwiająca zwolnienie procesora przez aktualny proces.

2.32.8. struct Registers

Struktura reprezentująca stan rejestrów kluczowych z punktu widzenia procesu w systemie operacyjnym.

Pola:

- uint32_t eax,
- uint32_t ebx,
- uint32_t ecx,
- uint32_t edx,
- uint32_t esi,
- uint32_t edi,
- uint32_t esp,
- uint32_t ebp,
- uint32_t eip,
- uint32_t eflags,
- uint32_t cr3.

2.32.9. struct Task

Struktura reprezentująca pojedynczy proces w systemie operacyjnym.

Pola:

- Registers registers - stan rejestrów;
- uint32_t isMain - informacja o tym czy proces jest procesem głównym systemu operacyjnego;
- uint32_t pid - liczba porządkowa przyporządkowana procesowi;
- void (*fun)() - funkcja do wykonania;
- struct Task *next - następne zadanie na liście;
- struct Task *previous - poprzednie zadanie na liście;