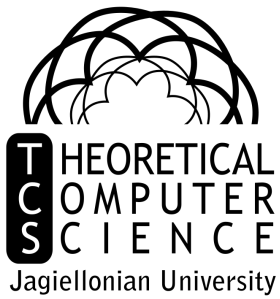
Programowanie Niskopoziomowe

semestr letni 2010/2011

Jakub Kozik

Informatyka Analityczna
tcs@jagiellonian



Zasady

Punkty

- 2 sprawdziany (po 20 punktów)
- 10 obowiązkowych zadań programistycznych submitowanych przez Aθinę/Satori (po 4 punkty)
- egzamin pisemny (20 punktów)

Ostateczna ocena

0-50	ndst
51-60	dst
61-70	+dst
71-80	db
81-90	+db
91-100	bdb

Program Wykładu

Część 1: programowanie aplikacji

- podstawy architektur x86 i x86_64
- programowanie w asemblerze (nasm)
- interface systemu Linux
- interface C/C++

Część 2: wydajność

- optymalizacja kodu
- self-modifying-code
- SIMD
- programowanie równoległe

Część 3: programowanie systemowe

- boot sequence
- stronicowanie
- segmentacja
- obsługa urządzeń

Literatura

Randall Hyde, The Art of Assembly Language, dostępna w intenecie

AMD64 Architecture Programmer's Manual

- Volume 1: Application Programming
- Volume 3: General-Purpose and System Instructions
- Volume 4: 128-bit and 256 bit media instructions
- Volume 2: System Programming

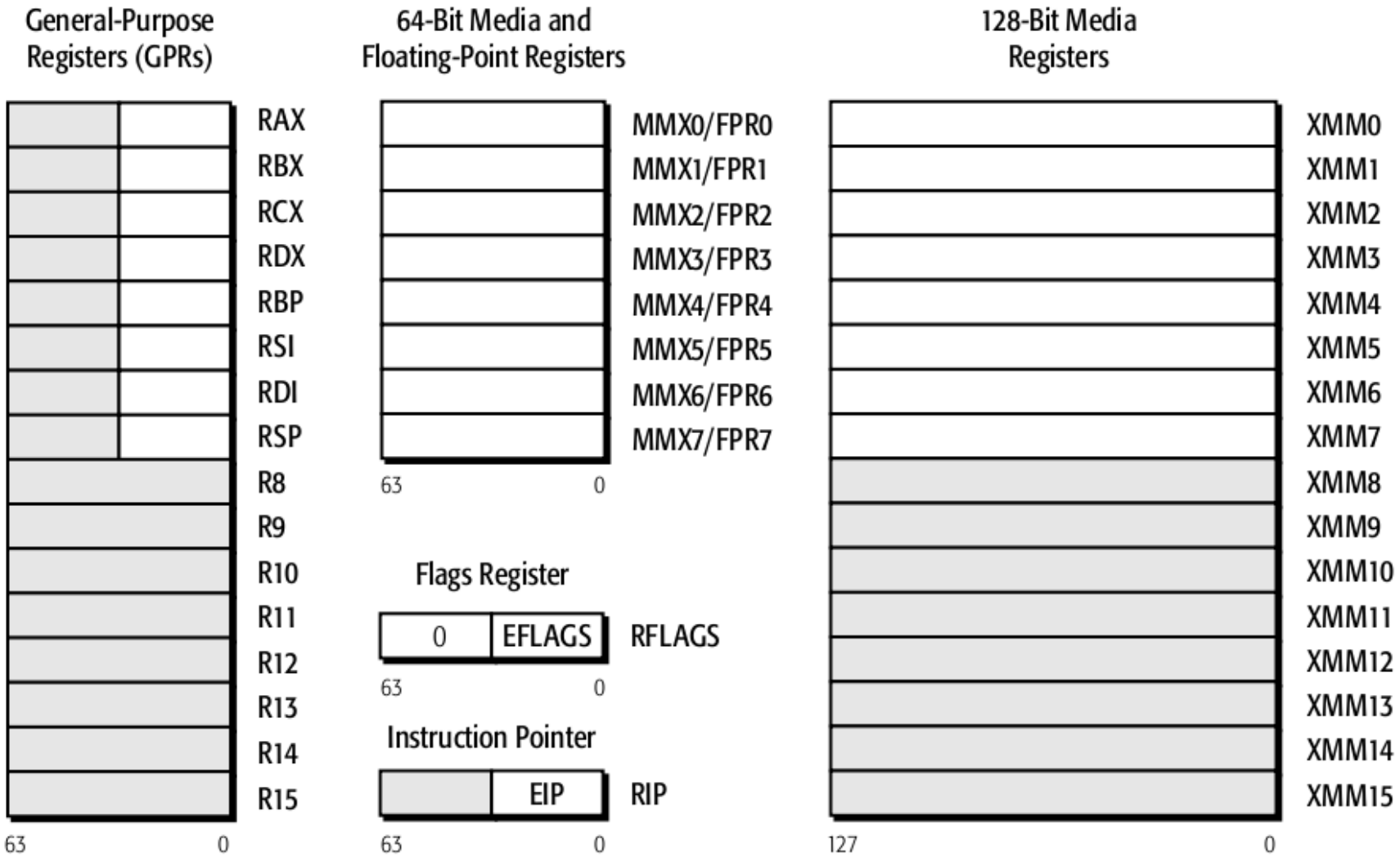
NASM - dokumetacja

System V Application Binary Interface
AMD64 Architecture Processor Supplement

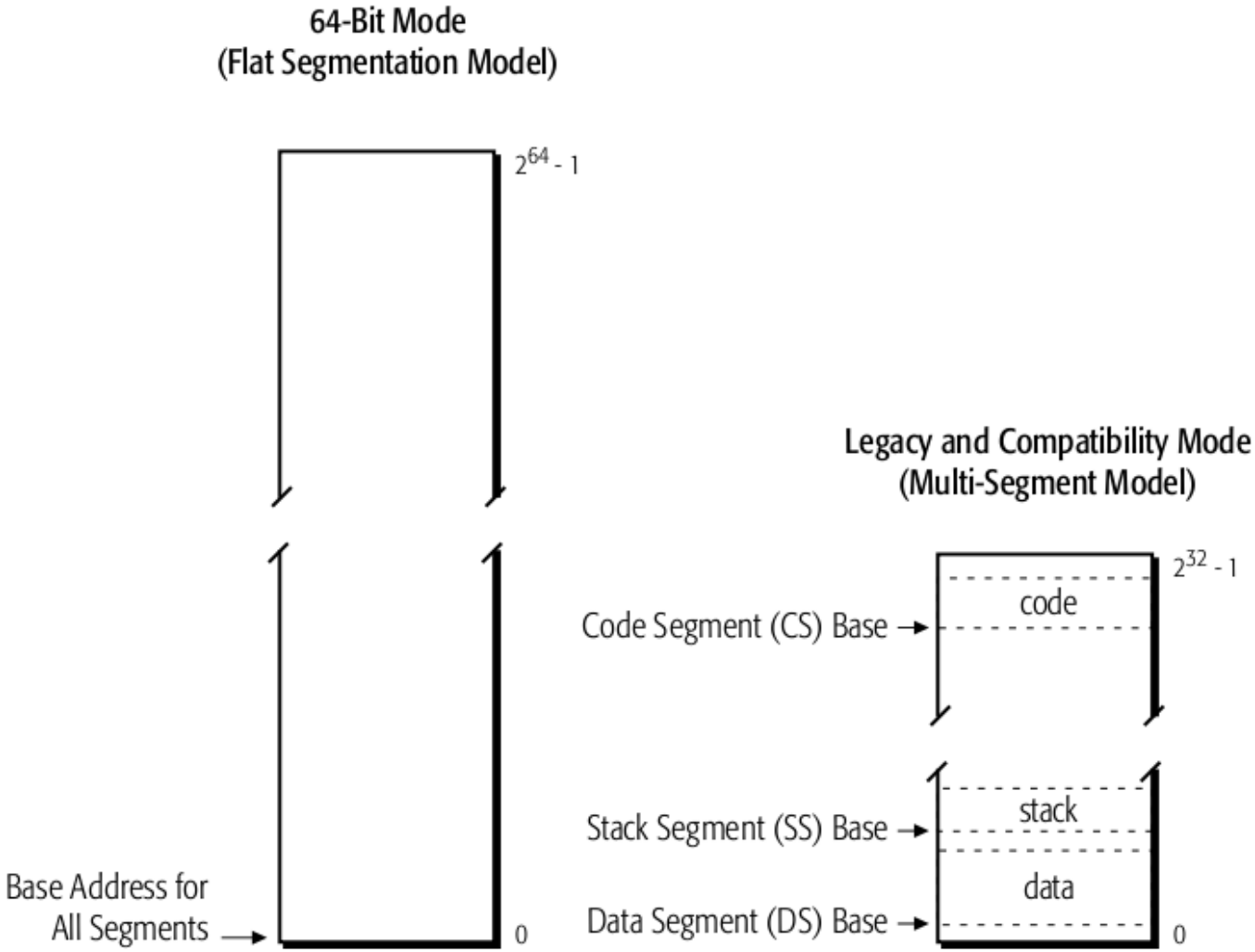
Tryby pracy procesora

- Legacy Mode
  - Protected Mode (32b) - segmentacja, stronicowanie, chroniona pamięć, 4GB pamięci
  - Real Mode (16b) - prosta segmentacja, pamięć nie chroniona, 1MB pamięci
  - Virtual-8086 (16b) - uruchamianie programów Real-mode w trybie Protected
- Long Mode
  - 64-Bit Mode (64b) - brak segmentacji, stronicowanie, chroniona pamięć, rejestry/adresowanie 64b
  - Compatibility Mode (32b/64b) - dla użytkownika wygląda jak Protected Mode, dla systemu jak 64-Bit

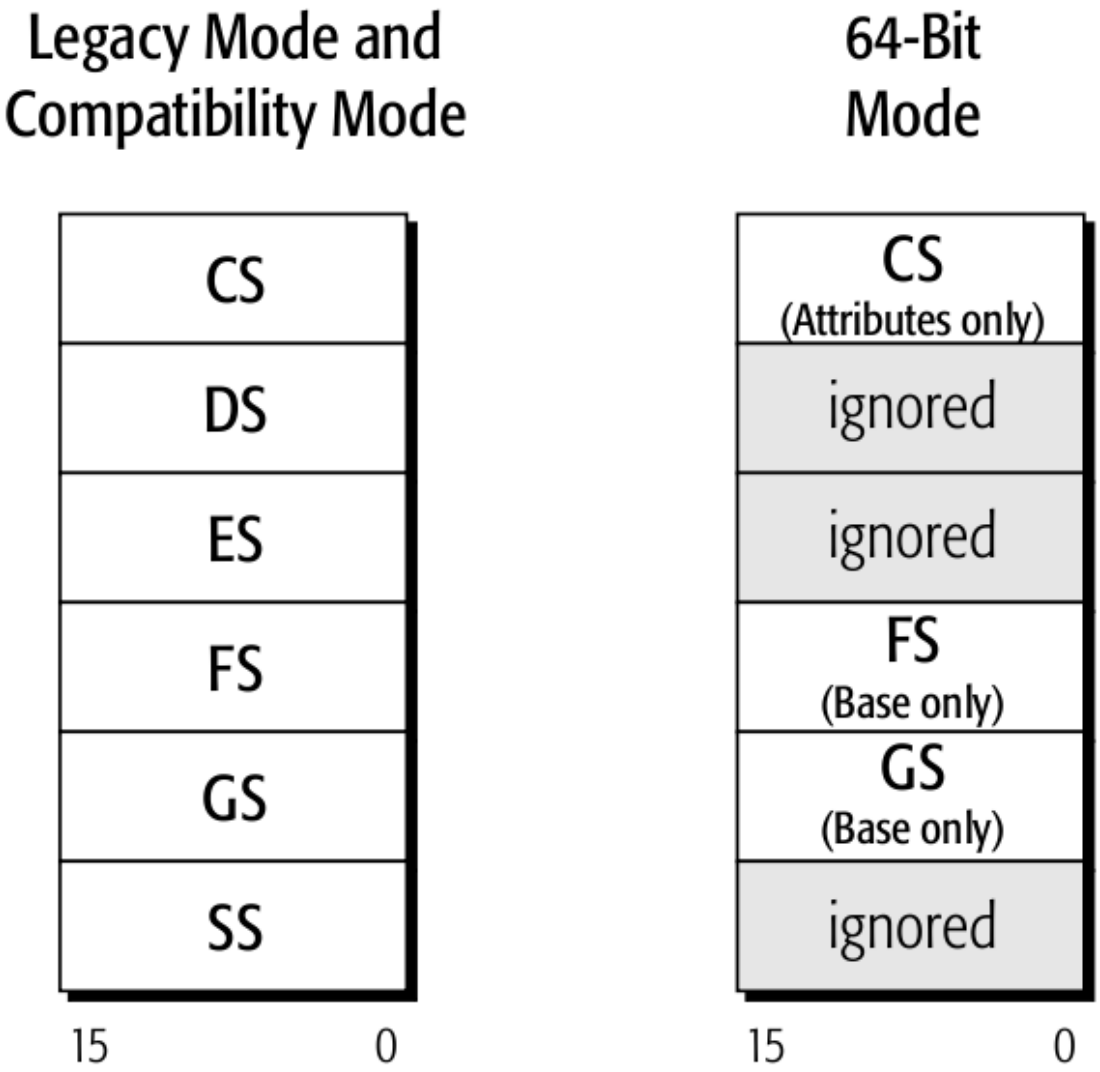
x86_64 rejestry użytkownika (General Purpose)



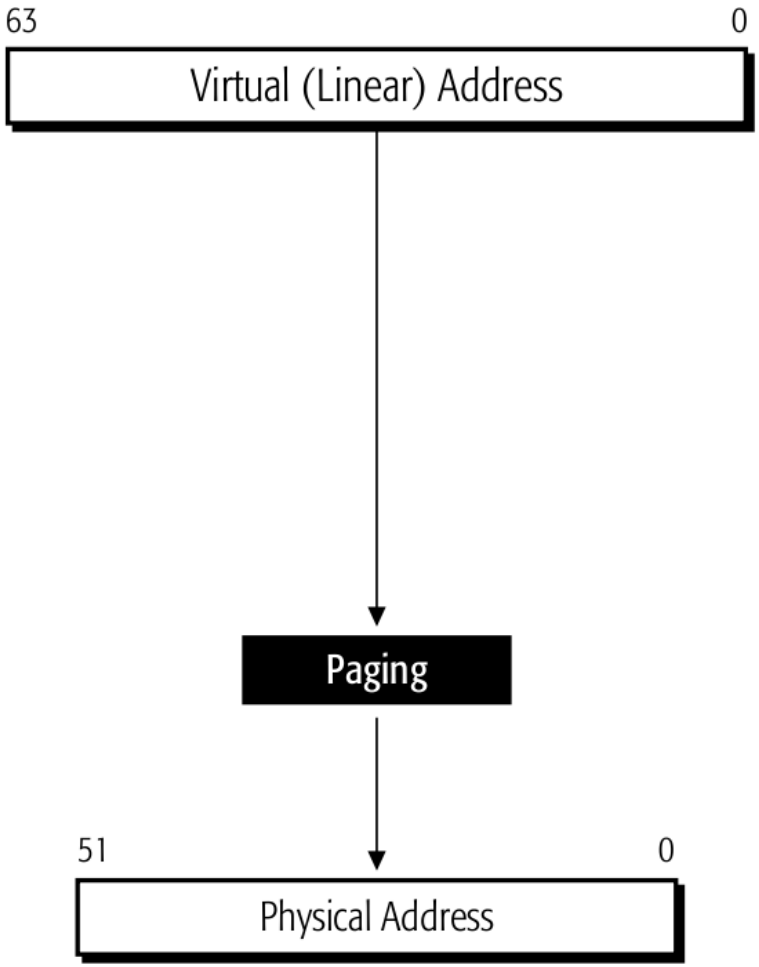
x86_64 organizacja pamięci

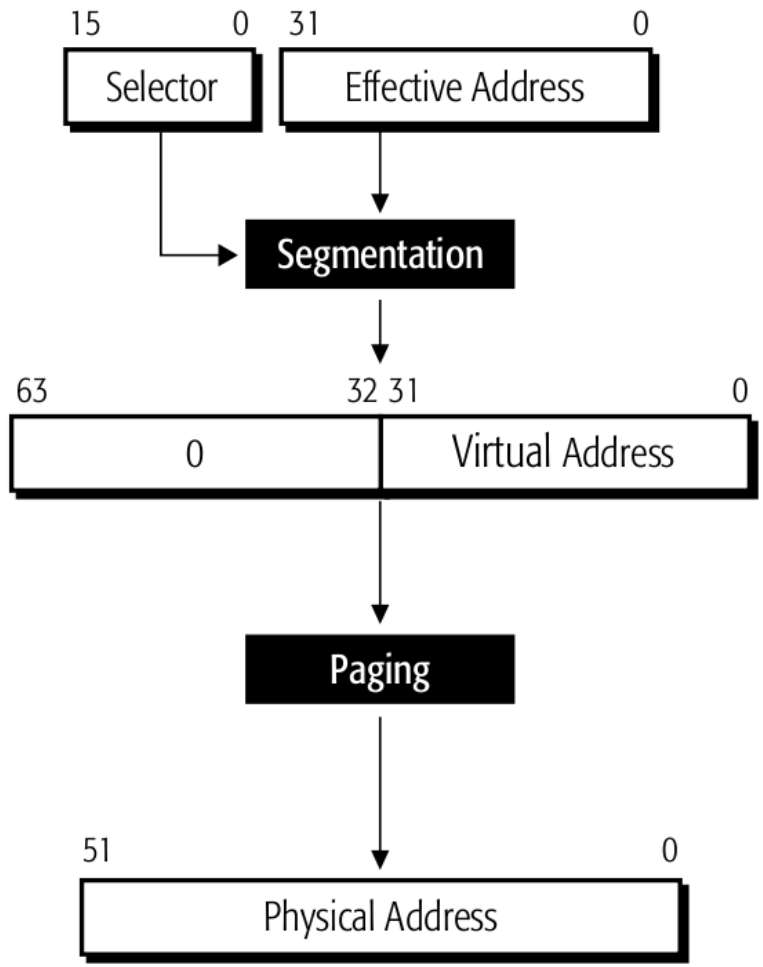


rejestry segmentów

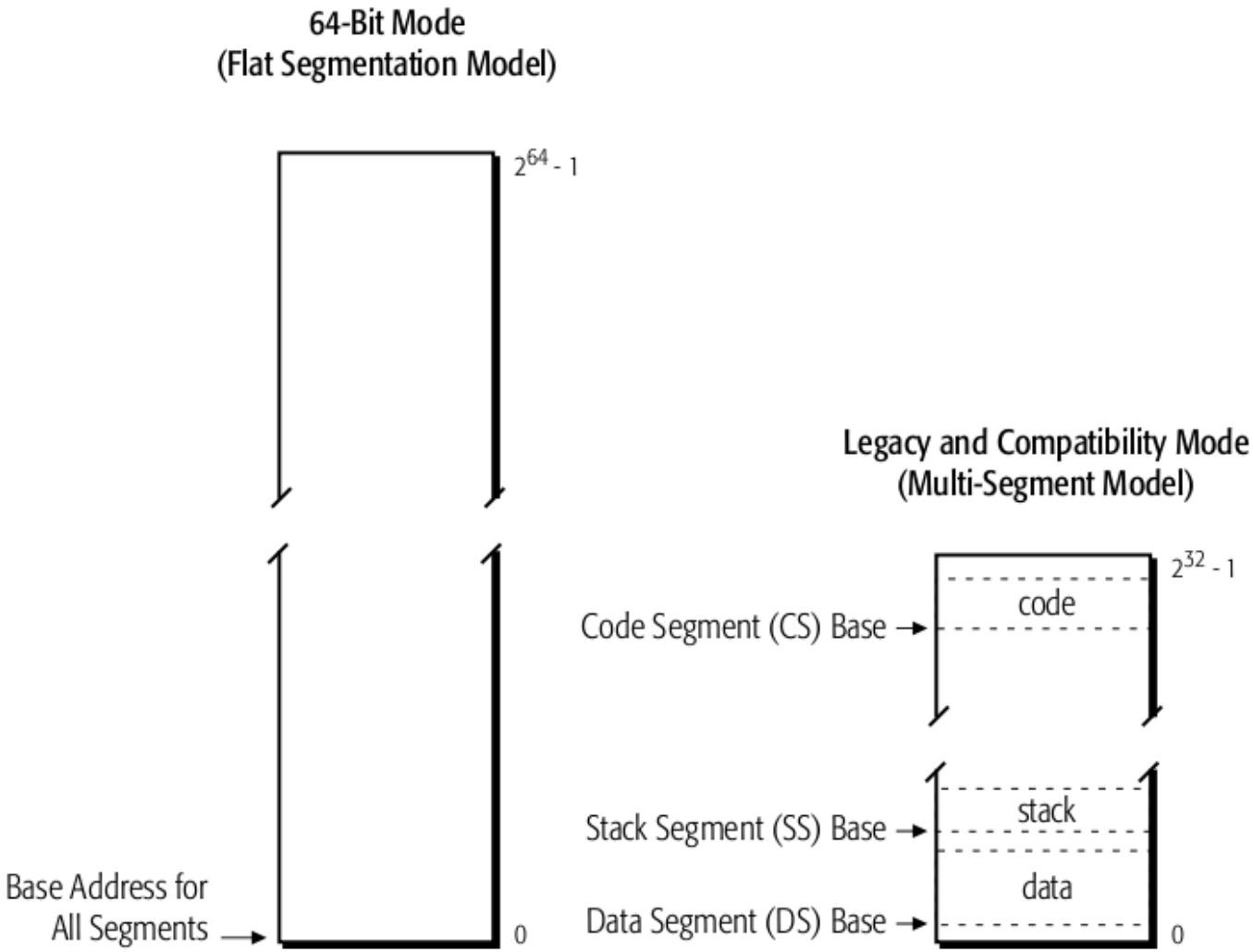


dostęp do pamięci





dostęp do pamięci

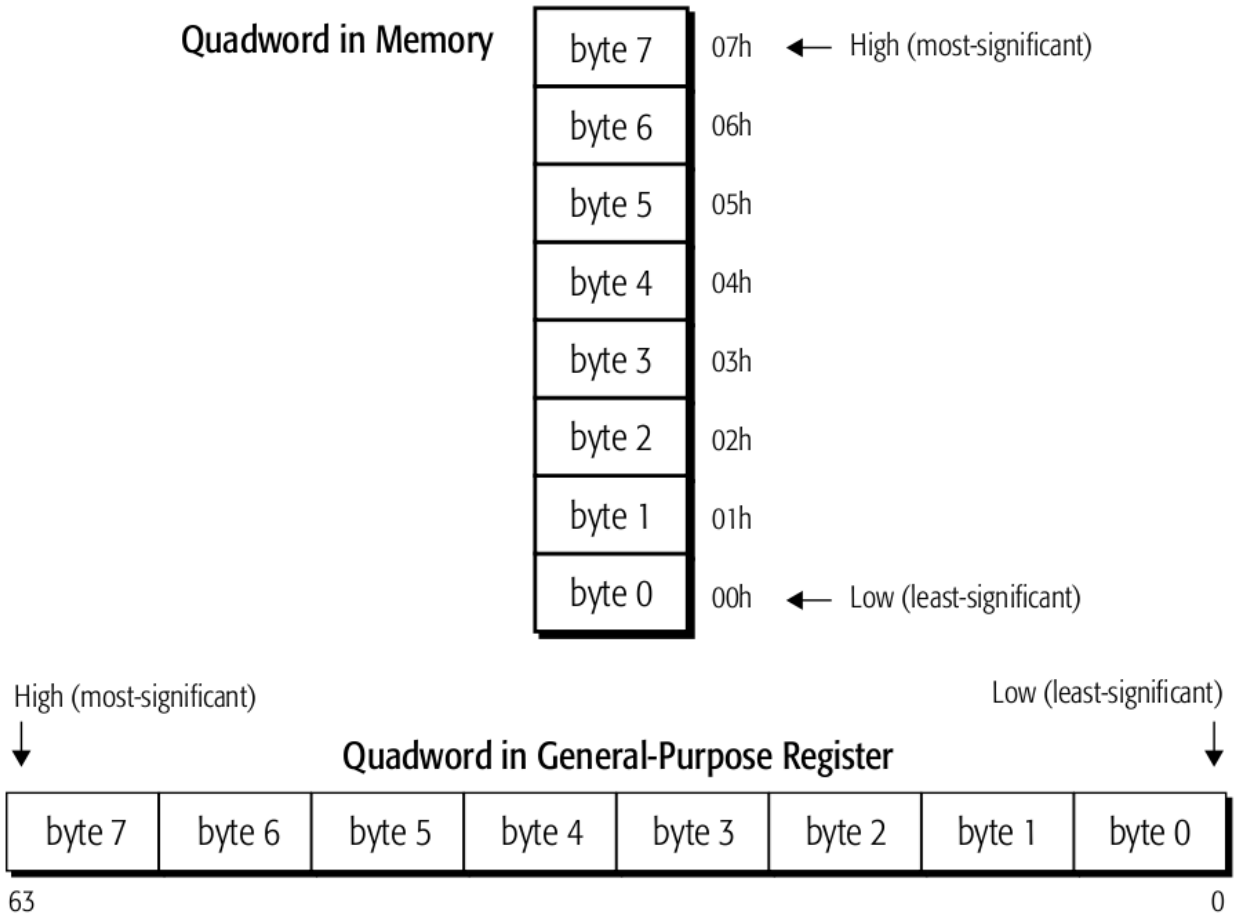


far/near pointers

Little/Big endian

- Little endian - najpierw najmniej znaczący byte
  - x86, x86_64
  - ARM -Linux
  - PowerPC - Solaris
- Big endian - najpierw najbardziej znaczący byte
  - SPARC - Linux
  - Microsoft Xbox 360, PlayStation 3, Nintendo Wii
  - PowerPC - Linux
  - TCP

Edianness


Quadword in Memory
byte 7
07h
High (most-significant)
byte 6
06h
byte 5
05h
byte 4
04h
byte 3
03h
byte 2
02h
byte 1
01h
byte 0
00h
Low (least-significant)
High (most-significant)
Low (least-significant)
Quadword in General-Purpose Register
byte 7
byte 6
byte 5
byte 4
byte 3
byte 2
byte 1
byte 0
63
0

Rodzaje adresowania

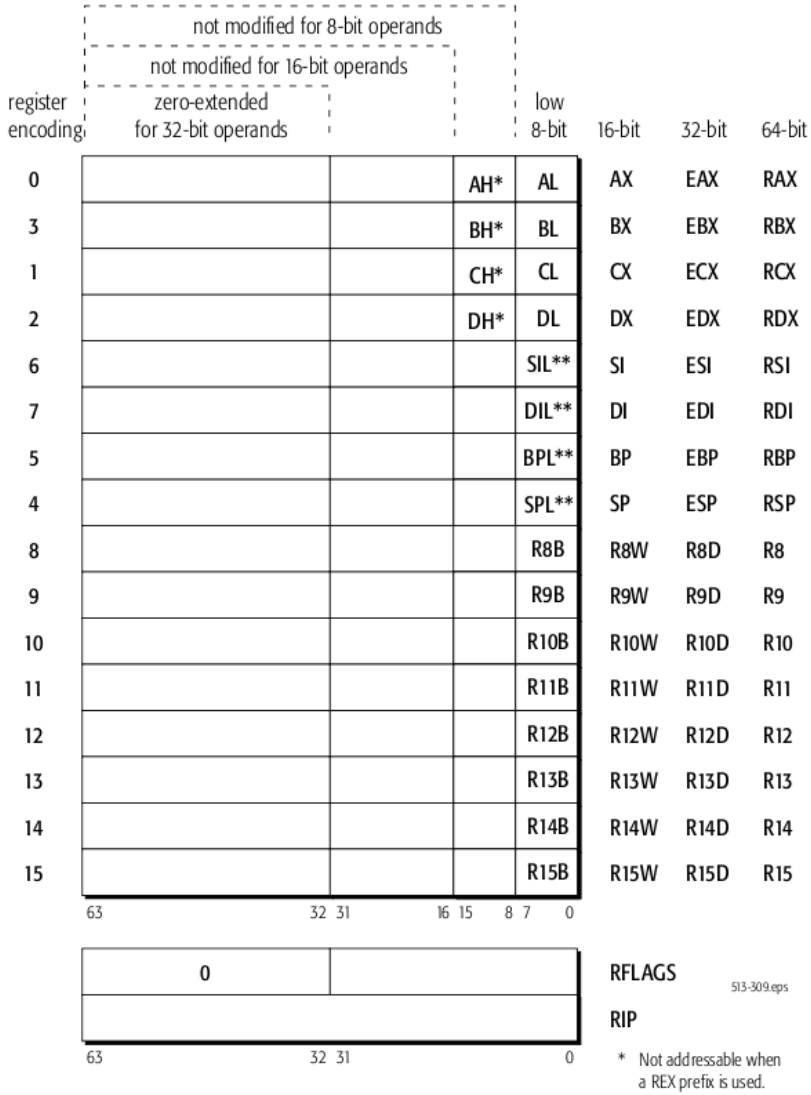
- Abolute Addresses
- ModR/M Addresses

$$Base + Index * Scale + Displacement$$

Base i *Index* w rejestrach, *Scale* oraz *Diplacement* w instrukcji. $Scale \in \{0, 1, 2, 4, 8\}$.

- Instruction-Relative Addresses
- Implicit Addresses(stos)
- String Addresses

Nazwy rejestrów GPR



Przykładowe wykorzystanie rejestrów

RAX akumulator

RCX licznik w pętlach

RDX wynik mnożenia MUL w RDX:RAX

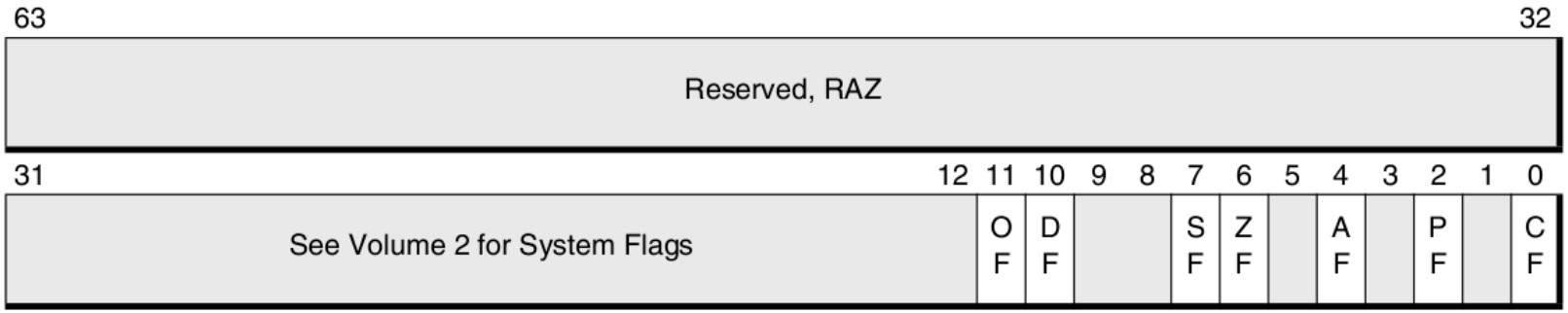
RSI adres źródła dla instrukcji typu *string*

RDI adres docelowy dla instrukcji typu *string*

RBP adres ramki stosu

RSP adres szczytu stosu

Rejestr flag



Bits	Mnemonic	Description	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
4	AF	Auxiliary Carry Flag	R/W
2	PF	Parity Flag	R/W
0	CF	Carry Flag	R/W

przykład - CMP

`CMP A B`

dane intepretowane jako `unsigned`

	CF	ZF
A $>$ B	0	0
A $=$ B	0	1
A $<$ B	1	0

dane intepretowane jako `signed`

	OF	ZF
A $>$ B	SF	0
A $=$ B	0	1
A $<$ B	NOT SF	0

Typy/Interpretacje danych

- `signed integers`
- `unsigned integers`
- `BCD digits` (Binary Coded Decimal)
- `Packed BCD digits`
- `strings`

Rozmiary danych

byte(8b), word(16b), doubleword (32b), quadword(64b), double quadword (128b)

Rodzaje operandów dla instrukcji

- rejestr
- adres w pamięci
- port I/O
- stała (`Immediate Operand`)

Dostęp do pamięci - szyna 64b

Konsekwencje:

- data alignment,
- im większe transfery tym lepsza wydajność.

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Podstawowe intrukcje

Data Transfer

MOV, CMOVcc, PUSH, POP,...

Arytmetyczne

ADD, SUB, NEG, MUL, IMUL, DIV, IDIV, DEC, INC, ... (+ LEA)

Testowanie i porównania

CMP, TEST, BT, SETcc, ...

Logiczne/Bitowe

AND, OR, XOR, NOT

Sterowanie przepływem

JMP, Jcc, LOOPcc, CALL, RET, ...

System Calls

SYSCALL (+SYSRET), SYSENTER (+SYSEXIT)

Stos

Stos - LIFO rośnie w dół, RSP wsakzuje na koniec stosu
Operacje:

- `push x`

```
sub rsp, sizeof(x)
mov [rsp], x
```

- `pop x`

```
mov x, [rsp]
sub rsp, sizeof(x)
```

- `call f`

```
push <next-instruction's-rip>
jmp f
```

- `ret`

```
pop rip
```

Przykład

$$f(n) = (x \mapsto 3x + 1)^{(n)}(0)$$

argument w EDI, wynik w EAX

Intel

```
<f>:
400544:   mov    eax,0x0
400549:   mov    edx,0x1
40054e:   test   edi,edi
400550:   jle    40055d <f+0x19>
400552:   lea    eax,[rax+rax*2+0x1]
400556:   add    edx,0x1
400559:   cmp    edi,edx
40055b:   jge    400552 <f+0xe>
40055d:   repz ret
```

AT&T

```
<f>:
400544:   mov  $0x0,%eax
400549:   mov  $0x1,%edx
40054e:   test %edi,%edi
400550:   jle  40055d <f+0x19>
400552:   lea  0x1(%rax,%rax,2),%eax
400556:   add  $0x1,%edx
400559:   cmp  %edx,%edi
40055b:   jge  400552 <f+0xe>
40055d:   repz retq
```

Kernel calls - MINIX

Wywołania systemowe

SENDREC

SEND, RECEIVE, NOTIFY

Kernel trap

```
push   ebp
mov ebp, esp
push   ebx
mov eax, SRC_DST(ebp)  ! eax = dest-src
mov ebx, MESSAGE(ebp)  ! ebx = message pointer
mov ecx, {SEND|RECEIVE|SENDREC|NOTIFY}
int SYSVEC             ! 33
pop ebx
pop ebp
ret
```

Kernel calls - Linux x86_64

Wywołania systemowe

- `exit`
- `open, close`
- `read, write`
- `brk`
- `mmap, munmap, mprotect`
- `fork, exec, wait`
- ...

Argumenty dla wywołań systemowych

`syscall`

- numer wywyłania systemowego w `rax`
- argumenty przekazywane są kolejno w rejestrach: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`
- argumentami mogą być liczby całkowite lub adresy w pamięci
- wywołanie może niszczyć zawartość rejestrów `rcx`, `r11`
- żadne argumenty nie są przekazywane przez stos
- wynik wywołania znajduje się w `rax`
  (wartości od -4095 do -1 oznaczają błąd (`-errno`))

Przykład

```
BITS 64
SECTION .text

GLOBAL _start
_start:
    mov rax, 1   ; __NR_write
    mov rdi, 1   ; fd (stdout)
    mov rsi, msg  ; *buf
    mov rdx, 7   ; count
    syscall

    mov rax, 60  ; __NR_exit
    mov rdi, 0   ; status

    syscall

SECTION .data
msg:
    db "Hello",10
```

Tabela (fragment)

syscall	rax	argumenty (rdi, rsi, ...)
read	0	INTEGER fd, void * buf, INTEGER count
write	1	INTEGER fd, void * buf, INTEGER count
open	2	void * pathname, INTEGER flags, INTEGER mode
close	3	INTEGER fd
stat	4	void * path, struct stat * buf
...	...	...
exit	60	INTEGER status
...	...	...

Numery wywołań systemowych

/usr/include/asm/unistd_64.h

stat (man page)

```
int stat(const char *path, struct stat *buf);
```

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

stat

nazwa	rozmiar	offset
st_dev	8	0
st_ino	8	8
st_mode	4	24
st_nlink	8	16
st_uid	4	28
st_gid	4	32
st_rdev	8	40
st_size	8	48
st_blksize	8	56
st_blocks	8	64
st_atime	8	72
st_mtime	8	88
st_ctime	8	104

```
sizeof(struct stat) = 144
```

Data alignment - 64bits (GNU, Intel Linux)

Wyrównanie w obrębie struktur

bool	char	short	int	long	long long	void *
1	1	2	4	8	8	8

float	double	long double (80bits)
4	8	16

Początkowy stan stosu procesu.

zawartość	adres	długość
argumenty, zmienne śr., aux inf.		
nieokreślone		
Null auxiliary vector entry		8b *2
Auxiliary vector entries		8b *2 (każdy)
0		8b
envp		8b (każdy)
0	rsp+8+8*argc	8b
argv	rsp+8	8b * argc
argc	rsp	8b

```
typedef struct{
  uint64_t a_type;     /* Entry type */
  union{
      uint64_t a_val;    /* Integer value */
  /*  void * a_ptr;
      void (*a_fnc)(); */
  } a_un;
} Elf64_auxv_t;
```

Auxiliary vector entries (ELF64)

Przykłady:

AT_PAGESX (6) rozmiar stron

AT_UID (11) identyfikator użytkownika (real uid)

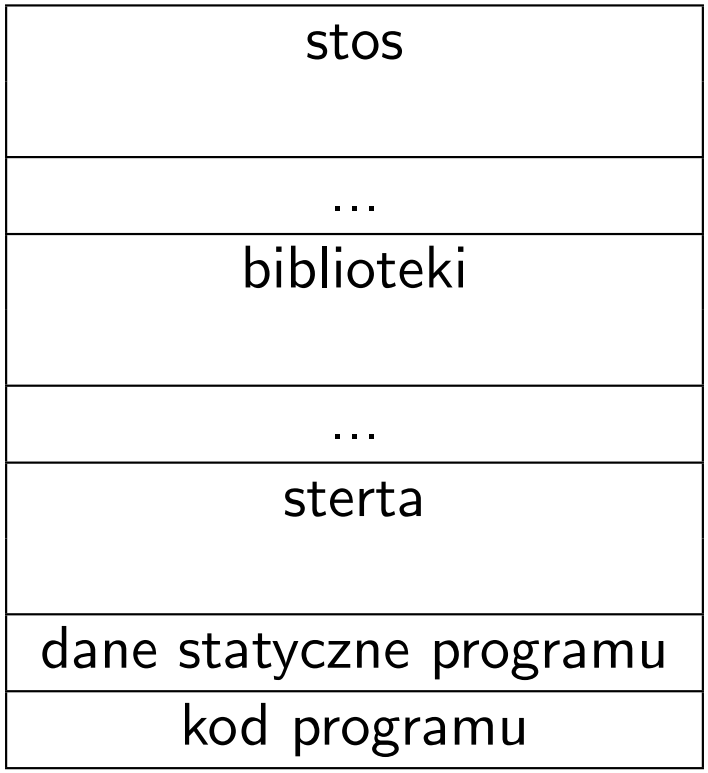
AT_EUID (12) efektywny identyfikator użutkownika (effective uid)

AT_GID (13) grupa użytkownika (real gid)

AT_EGID (14) efektywna grupa użutkownika (effective gid)

...

Obraz pamięci procesu.



Obraz pamięci procesu.

```
cat /proc/self/maps
```

```
00400000-0040d000 r-xp 00000000 08:05 262273                 /bin/cat
0060d000-0060e000 r--p 0000d000 08:05 262273                 /bin/cat
0060e000-0060f000 rw-p 0000e000 08:05 262273                 /bin/cat
00ecd000-00eee000 rw-p 00000000 00:00 0                      [heap]
7fce156f2000-7fce1586c000 r-xp 00000000 08:05 1314530    /lib/libc-2.11.1.so
...
7fce15c94000-7fce15c95000 r--p 0001f000 08:05 1311478    /lib/ld-2.11.1.so
7fce15c95000-7fce15c96000 rw-p 00020000 08:05 1311478    /lib/ld-2.11.1.so
7fce15c96000-7fce15c97000 rw-p 00000000 00:00 0
7fff0f895000-7fff0f8aa000 rw-p 00000000 00:00 0          [stack]
7fff0f9ff000-7fff0fa00000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Statyczna alokacja pamięci

Segment kodu

```
SECTION .text
napis:  db "halo"
```

Segment danych inicjalizowanych

```
SECTION .data
zmienna:  dw 42
```

Segment danych nieinicjalizowanych

```
SECTION .bss
bufor:  resb 4096
```

Statyczna alokacja pamięci

Segment kodu

```
SECTION .text
napis:  db "halo"
```

Segment danych inicjalizowanych

```
SECTION .data
zmienna:  dw 42
```

Segment danych nieinicjalizowanych

```
SECTION .bss
bufor:  resb 4096
```

Statyczna alokacja pamięci

Segment kodu

```
SECTION .text
napis:  db "halo"
```

Segment danych inicjalizowanych

```
SECTION .data
zmienna:  dw 42
```

Segment danych nieinicjalizowanych

```
SECTION .bss
bufor:  resb 4096
```

Dynamiczna alokacja pamięci

stos

`brk`

- przesuwa granicę dostępnego obszaru dynamicznego (`heap`)
- obszar ten jest zawsze spójny
- proces musi sam nim zarządzać (`malloc`)

`mmap`

- mapuje plik do przestrzeni adresowej procesu
- zmapowany obszar może być współdzielony
- za pomocą `mmap` można też żądać nowych stron pamięci bez wskazywania pliku
- zawsze całokowita liczba stron

Dynamiczna alokacja pamięci

stos

brk

- przesuwa granicę dostępnego obszaru dynamicznego (heap)
- obszar ten jest zawsze spójny
- proces musi sam nim zarządzać (malloc)

mmap

- mapuje plik do przestrzeni adresowej procesu
- zmapowany obszar może być współdzielony
- za pomocą mmap można też żądać nowych stron pamięci bez wskazywania pliku
- zawsze całokowita liczba stron

Dynamiczna alokacja pamięci

stos

`brk`

- przesuwa granicę dostępnego obszaru dynamicznego (`heap`)
- obszar ten jest zawsze spójny
- proces musi sam nim zarządzać (`malloc`)

`mmap`

- mapuje plik do przestrzeni adresowej procesu
- zmapowany obszar może być współdzielony
- za pomocą `mmap` można też żądać nowych stron pamięci bez wskazywania pliku
- zawsze całokowita liczba stron

vsyscall

vsyscall

```
ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked
```

funkcje

- gettimeofday
- clock_gettime
- getcpu

Kernel calls - Linux x86

`int 0x80`

- numer wywyłania systemowego w `eax`
- argumenty przekazywane są kolejno w rejestrach: `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`
- argumentami mogą być liczby całkowite lub adresy w pamięci
- wynik wywołania znajduje się w `eax`

sysenter

sysenter

Bardziej wydajny, ale ...

... gubi `eip, esp`.

Rozwiązanie: *sysenter trampoline* w ustalonym miejscu.
Dostęp przez bibliotekę(stronę) systemową `vdso`.

Adres `vdso` nie jest stały. Możemy go odczytać z `Auxiliary vector entries`.

```
auxv->a_type == AT_SYSINFO_EHDR
```

Numery wywołań systemowych (32 bity)

syscall	numer (32 bity)	numer (64 bity)
exit	1	60
fork	2	57
read	3	0
wite	4	1
open	5	2
close	6	3
...	...	...

Jeszcze jeden przykład

```
int getdents(unsigned int fd, struct linux_dirent *dirp,
             unsigned int count);
```

```
struct linux_dirent {
    unsigned long  d_ino;     /* Inode number */
    unsigned long  d_off;     /* Offset to next linux_dirent */
    unsigned short d_reclen;  /* Length of this linux_dirent */
    char           d_name[];  /* Filename (null-terminated) */
                       /* length is actually (d_reclen - 2 -
                          offsetof(struct linux_dirent, d_name) *
    /*
    char pad;       // Zero padding byte
    char d_type;    // File type (only since Linux 2.6.4;
                               // offset is (d_reclen - 1))
    */
}
```

Data alignment - zmienne

Zmienne

Nie muszą być wyrównywane, ale warto wyrównywać ze względu na wydajność.

Zmienne - wyjątek

Zmienne typu `packed` dla operacji SIMD muszą być wyrównane:

__m64 (MMX)	__m128 (SSE)	__m256 (AVX)
8	16	32

Data alignment - struktury

Wyrównanie w obrębie struktur

bool	char	short	int	long	long long	void *
1	1	2	4	8	8	8

float	double	long double (80bits)
4	8	16

Dodatkowo

- rozmiar obiektu/struktury jest zawsze wielokronością wyrównań zmiennych składowych
- tablice o rozmiarze przynajmniej 16 bajtów muszą być wyrównane do 16 bajtów
- `variable-length arrays` muszą być wyrównane do 16 bajtów

Wywoływanie funkcji

Przynależność rejestrów

Wywołanie funkcji musi zachowywać wartości rejestrów:
`rbp, rbx, ,r12-r15`
Wszystkie pozostałe mogą być zmienione.

Flagi

- `DF` (Direction Flag) powinna być skasowana (0) przy wchodzeniu i wychodzeniu z funkcji.
- Pozostałe flagi mogą być zmieniane.

- Bity konktrolne rejestru `MXCSR` muszą być zachowane.
- x87 control word musi być zachowane.

Ramka stosu

Pozycja	Zawartość	Przynależność
rbp+8n+16	n-ty argument na stosie	poprzednia
	...	
rbp+16	0-wy argument na stosie	
rbp+8	adres powrotny	bieżąca
rbp	poprzednia wartość rbp	
rbp-8	zmienne lokalne	
	...	
rsp	zmienne lokalne	
rsp-128	`red zone`	

W momencie wywołania funkcji stos powinien byc wyrównany do 16..

Przekazywanie argumentów

Rejestry używane do przekazywania argumentów

- GPR: `rdi, rsi, rdx, rcx, r8, r9`
- SSE: `xmm0, ... , xmm7`
- `al` (czasami)

Argumenty przekazywane przez stos są ułożone od prawego do lewego.

Klasyfikacja argumentów

INTEGER	przekazywane w GPR
SSE	przekazywane przez rejestry SSE (dolny quadword)
SSEUP	przekazywane przez rejestry SSE (górny quadword)
X87, X87UP	64b mantysa i 2b (+6b padding) wykładnik typu long double
MEMORY	przkazywane przez stos

Przekazywanie argumentów

Proste typy

- `bool, char, int, long, void *` - INTEGER
- `float, double, __m64` - SSE
- `__m128, __float128` - SEE, SSEUP
- `__m256` -SSE, SSEUP, SSEUP, SSEUP
- `long double` - X87, X87UP $\rightarrow$ MEMORY (128b) (stos)

Uwaga: `void f(...)`

W rejestrze `al` liczba argumentów przekazywanych przez SSE.

Przekazywanie argumentów

Struktury

- Maksymalny rozmiar struktury, który może być przekazany przez rejestry to 16B.
- Niewyrównane struktury są przekazywane przez stos.
- Każde 8B struktury jest klasyfikowane/przekazywane oddzielnie.
- Każdy element struktury jest klasyfikowany oddzielnie.
- Reguły łączenia elementów we wspólny typ dla 8B kawałków:
  - jeśli jakiś element jest typu MEMORY to całe 8B jest przekazywane przez stos
  - jeśli jakiś element jest typu INTEGER to całe 8B dostaje typ INTEGER
  - jeśli jakiś element jest typu X87, X87UP to całe 8B dostaje typ MEMORY
  - wpp 8B dostaje typ SSE
- Jeśli któryś z 8B struktury jest typu MEMORY to cała struktura jest przekazywana przez stos.

Wynik funkcji

- jeśli wynik jest typu MEMORY to wołający musi podać adres na zwracaną strukturę (jako pierwszy parametr - `rdi`). Po wyjsciu rax bedzie zawierał adres tej struktury.
- wynik typu INTEGER - `rax`
- wynik typu SSE, SSEUP - `xmm0`
- wynik typu X87, X87UP - `st0`
- wynik jest strukturą, która może być przekazana przez rejestry - używamy `rax, rdx, xmm0, xmm1`

Przykład

```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;
extern void func (int e, int f,
     structparm s, int g, int h,
     long double ld, double m,
     __m256 y,
     double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);
```

GPR	SSE	stos
rdi: e	xmm0: s.d	0: ld
rsi: f	xmm1: m	16: j
rdx: s.a, s.b	ymm2: y	24: k
rcx: g	xmm3: n	
r8: h		
r9: i		

Konwencja wołania funkcji UNIX 32b

Argumenty

- argumenty na stosie (od prawej do lewej)
- wynik w `al`, `ax`, `eax`, `edx:eax`

Przynależność rejestrów

- `ebx`, `ebp`, `esi`, `edi` - funkcja wywołująca
- pozostałe funkcja wywoływana

Statyczne łącznie modułów (nasm)

Nazwy funkcji takie jak w C.

Import symbolu do modułu assemblerowego:

```
EXTERN funkcja
call funkcja

EXTERN zmienna
mov rax, [zmienna]
```

Eksport symbolu z modułu assemblerowego:

```
GLOBAL funkcja
GLOBAL zmienna
```

Biblioteki współdzielone - Dynamic Shared Objects

- Segment kodu biblioteki dzielony przez wiele procesów.
- Ten sam kod ma różne adresy w przestrzeniach wirtualnych procesów.
- Sekcje danych są oddzielne dla procesów korzystających z biblioteki, ale również są w innych miejscach.

PIC

DSO muszą być implementowane jako Position Independen Code.

Position Independent Code (64b)

Adresowanie względne.

AMD64 - tryb adresowania względem `rip`

```
mov rax, [rel zmienna]
```

- Wszystkie sekcje biblioteki są ladowane w spójnym bloku pamięci.
- Do własnych danych możemy się odwoływać poprzez adresowanie względem `rip`.

DSO - exportowanie funkcji i zmiennych

funkcje

```
GLOBAL  func:function
```

zmienne

```
global  array:data array.end-array

array:  resd    128
.end:
```

Procedure Linkage Table

PLT

"Przekierowania" dla funkcji importowanych spoza biblioteki.

Docelowo dla każdej funkcji *stub* postaci:

```
printf:
jmp [adres printfa w pamięci wirtualnej bieżącego procesu]
```

Początkowo kod *dynamic-linkera*, który ładuje bibliotekę w razie potrzeby i uzupełnia referencje.

W kodzie biblioteki (nasm)

```
call printf wrt ..plt
```

Global Offset Table

DSO dane zewnętrzne (`EXTERN`)

- są w nieznanych miejscach pamięci
- niezależnych od `rip`
- dane publiczne (exportowane) traktowane są tak jak zewnętrzne - przestrzeń dla nich jest alokowana na stercie

Global Offset Table

- Tablica z adresami danych zewnętrznych,
- w sekcji danych bibliteki współdzielonej,
- uzupełniana przez *dynamic-linker*.

Referencja do danych poprzez GOT (nasm)

```
EXTERN ext_var
...
mov rax, [ext_var wrt ..got]
```

Position Independent Code 32b

Główne różnice

- Brak adresowania względem `rip`
- Sekcja danych może być w innym miejscu w pamięci.

Position Independent Code 32b

32b - sztuczka z odczytaniem eip

```
      call next
next: pop ebx
```

GOT

Przesunięcie adresu tablicy GOT względem początku sekcji kodu można odczytać ze zmiennej _GLOBAL_OFFSET_TABLE_ (EXTERN).

```
add     ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc
```

GOT

GOT jest w sekcji danych więc zmienne biblioteki możemy adresować względem GOT.

```
lea     ebx, [ebx+ myvar wrt ..gotoff]
```

Przykład

```c
int add(int a, int b){
  return a+b;
}

double add(double a, double b){
  return a+b;
}
```

gcc

```
add.c:5: error: conflicting types for ‘add’
add.c:1: note: previous definition of ‘add’ was here
```

g++

```
OK
```

Przykład

```
int add(int a, int b){
  return a+b;
}

double add(double a, double b){
  return a+b;
}
```

g++ - tabela syboli (fragment)

```
0000000000000000 g     F .text 0000000000000015 _Z3addii
0000000000000015 g     F .text 000000000000001a _Z3adddd
```

Name mangling

Jak unikać?

```
extern C int add(int, int);
```

Jak czytać nazwy?

```
<mangled-name> ::= _Z <encoding>
<encoding> ::= <function name> <bare-function-type>
       ::= <data name>
       ::= <special-name>
```

Name mangling - proste funkcje

```
<function name> ::= ... ::= <source-name>
                ::= ...

<source-name> ::= <positive length number> <identifier>
```

g++ - tabela symboli (fragment)

```
0000000000000000 g     F .text 0000000000000015 _Z3addii
0000000000000015 g     F .text 000000000000001a _Z3adddd
```

Name mangling - typy argumentów

Typy wbudowane

```
<builtin-type> ::= v # void
::= c # char
::= i # int
::= j # unsigned int
::= l # long
::= m # unsigned long
::= d # double
::= e # long double, __float80
::= ...
```

g++ - tabela symboli (fragment)

```
0000000000000000 g     F .text 0000000000000015 _Z3addii
0000000000000015 g     F .text 000000000000001a _Z3adddd
```

Zagnieżdżone definicje

```
class Foo {
public:
    int foo (double*);
    double bar (int, double* );
};

int Foo::foo(double* d) { //_ZN3Foo3fooEPd
    return 1;
}

double Foo::bar(int i, double*){ //_ZN3Foo3barEiPd
    return 0.9;
}
```

Jeszcze kilka przykładów

```
_Zrm1XS_                Ret? operator%(X, X);
_ZplR1XS0_              Ret? operator+(X&, X&);
_ZlsRK1XS1_             Ret? operator<< (X const&, X const&);
_Z3foo5Hello5WorldS0_S_ Type? foo(Hello,World,World,Hello)
```

Jak sprawdzać nazwy?

```
objdump -t file.o
```

Wywoływanie funkcji/metod

Referencje

Itnieją tylko na poziomie składni/kontroli typów. W kodzie maszynowym to normalene wskaźniki.

Metody obiektów

Zwykłe funkcje, których pierwszy argument to wskaźnik `this`.

Enkapsulacja (public/private/protected

Istnieje tylko na poziomie kodu C++.

Wywoływanie funkcji/metod

Referencje

Itnieją tylko na poziomie składni/kontroli typów. W kodzie maszynowym to normalene wskaźniki.

Metody obiektów

Zwykłe funkcje, których pierwszy argument to wskaźnik `this`.

Enkapsulacja (public/private/protected

Istnieje tylko na poziomie kodu C++.

Wywoływanie funkcji/metod

Referencje

Itnieją tylko na poziomie składni/kontroli typów. W kodzie maszynowym to normalene wskaźniki.

Metody obiektów

Zwykłe funkcje, których pierwszy argument to wskaźnik `this`.

Enkapsulacja (public/private/protected

Istnieje tylko na poziomie kodu C++.

Proste klasy

```
class A {
public:
  long a;
  void fA();
};

class B : public A {
public:
  long b;
  void fB();
};
```

Metody, które nie są wirtualne isntnieją jako funkcje niezależne od obiektów.

Proste klasy

```
class A {
public:
  long a;
  void fA();
};

class B : public A {
public:
  long b;
  void fB();
};
```

Metody, które nie są wirtualne isntnieją jako funkcje niezależne od obiektów.

Proste klasy

```
class A {
public:
  long a;
  void fA();
};

class B : public A {
public:
  long b;
  void fB();
};
```

Metody, które nie są wirtualne isntnieją jako funkcje niezależne od obiektów.

Proste klasy

```
class A {
public:
  long a;
  void fA();
};

class B : public A {
public:
  long b;
  void fB();
};
```

Metody, które nie są wirtualne isntnieją jako funkcje niezależne od obiektów.

Proste klasy

```
class A {
public:
  long a;
};

class B : public A {
public:
  long b;
};
```

A w pamięci

offset	value
+0	a

B w pamięci

offset	value
+0	a
+8	b

Proste klasy

```
class A {
public:
  long a;
};

class B : public A {
public:
  long b;
};
```

A w pamięci

offset	value
+0	a

B w pamięci

offset	value
+0	a
+8	b

Proste klasy

```
class A {
public:
  long a;
};

class B : public A {
public:
  long b;
};
```

A w pamięci

offset	value
+0	a

B w pamięci

offset	value
+0	a
+8	b

Proste klasy

```
class A {
public:
  long a;
};

class B : public A {
public:
  long b;
};
```

A w pamięci

offset	value
+0	a

B w pamięci

offset	value
+0	a
+8	b

Proste klasy

```
class A {
public:
  long a;
};

class B : public A {
public:
  long b;
};
```

A w pamięci

offset	value
+0	a

B w pamięci

offset	value
+0	a
+8	b

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Metody wirtualne

```
class A {
public:
  long a;
  virtual void v();
};

class B {
public:
  long b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→typeinfo (A)
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→B::w()

VTable

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

vtableA

offset	value
+0	0 (top_offset)
+8	→`typeinfo (A)`
+16	→A::v()

vtableC

offset	value
+0	0 (top_offset)
+8	→`typeinfo (C)`
+16	→A::v()
+24	-16 (top_offset)
+32	→`typeinfo (C)`
+40	→B::w()

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

Przykryta metoda wirtualna

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

A w pamięci

offset	value
+0	→ vtableA+16
+8	a

B w pamięci

offset	value
+0	→ vtableB+16
+8	b

C w pamięci

offset	value
+0	→ vtableC+16
+8	a
+16	→ vtableC+48
+24	b
+32	c

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

```
this -= 16 (przesuń wskaźnik)
skocz do C::w()
```

Uwaga! top_offset to przesunięcie w obiekcie, a nie w `vtable`.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→`typeinfo (C)`
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→`typeinfo (C)`
+40	→thunk dla B::w()

thunk dla B::w()

- `this`-=16 (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w `vtable`.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- this-=16 (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w vtable.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- this-=16 (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w vtable.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- this-=16 (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w vtable.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- `this-=16` (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w `vtable`.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- `this-=16` (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w `vtable`.

VTable

```
class A {
public:
  int a;
  virtual void v();
};

class B {
public:
  int b;
  virtual void w();
};

class C : public A, public B {
public:
  int c;
  void w();
};
```

vtableC

offset	value
+0	0 (top_offset)
+8	→typeinfo (C)
+16	→A::v()
+16	→C::v()
+24	-16 (top_offset)
+32	→typeinfo (C)
+40	→thunk dla B::w()

thunk dla B::w()

- `this-=16` (przesuń o top_offset)
- skocz do C::w()

Uwaga! top_offset to przesunięcie w obiekcie, a nie w `vtable`.

Wielokrotne dziedziczenie

```
class A {
public:
    int a;
    virtual void v();
};

class B : public A {
public:
    int b;
    virtual void w();
};

class C : public A {
public:
    int c;
    virtual void x();
};

class D : public B, public C {
public:
    int d;
    virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

Wielokrotne dziedziczenie

```
class A {
public:
    int a;
    virtual void v();
};

class B : public A {
public:
    int b;
    virtual void w();
};

class C : public A {
public:
    int c;
    virtual void x();
};

class D : public B, public C {
public:
    int d;
    virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

Wielokrotne dziedziczenie

```
class A {
public:
    int a;
    virtual void v();
};

class B : public A {
public:
    int b;
    virtual void w();
};

class C : public A {
public:
    int c;
    virtual void x();
};

class D : public B, public C {
public:
    int d;
    virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

Wielokrotne dziedziczenie

```
class A {
public:
    int a;
    virtual void v();
};

class B : public A {
public:
    int b;
    virtual void w();
};

class C : public A {
public:
    int c;
    virtual void x();
};

class D : public B, public C {
public:
    int d;
    virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

vtableD

offset	value
+0	0 (top_offset)
+8	→typeinfo (D)
+16	→A::v()
+24	→B::w()
+32	→D::y()
+40	-24 (top_offset)
+48	→typeinfo (D)
+56	→A::v()
+64	→C::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

vtableD

offset	value
+0	0 (top_offset)
+8	→typeinfo (D)
+16	→A::v()
+24	→B::w()
+32	→D::y()
+40	-24 (top_offset)
+48	→typeinfo (D)
+56	→A::v()
+64	→C::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

vtableD

offset	value
+0	0 (top_offset)
+8	→typeinfo (D)
+16	→A::v()
+24	→B::w()
+32	→D::y()
+40	-24 (top_offset)
+48	→typeinfo (D)
+56	→A::v()
+64	→C::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+16
+8	a
+16	b
+24	→ vtableD+56
+32	a
+40	c
+48	d

vtableD

offset	value
+0	0 (top_offset)
+8	→`typeinfo (D)`
+16	→A::v()
+24	→B::w()
+32	→D::y()
+40	-24 (top_offset)
+48	→`typeinfo (D)`
+56	→A::v()
+64	→C::v()

Wielokrotne dziedziczenie

```
class A {
public:
  int a;
  virtual void v();
};

class B : public virtual A {
public:
  int b;
  virtual void w();
};

class C : public virtual A {
public:
  int c;
  virtual void x();
};

class D : public B, public C {
public:
  int d;
  virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

Wielokrotne dziedziczenie

```
class A {
public:
  int a;
  virtual void v();
};

class B : public virtual A {
public:
  int b;
  virtual void w();
};

class C : public virtual A {
public:
  int c;
  virtual void x();
};

class D : public B, public C {
public:
  int d;
  virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

Wielokrotne dziedziczenie

```
class A {
public:
  int a;
  virtual void v();
};

class B : public virtual A {
public:
  int b;
  virtual void w();
};

class C : public virtual A {
public:
  int c;
  virtual void x();
};

class D : public B, public C {
public:
  int d;
  virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

Wielokrotne dziedziczenie

```
class A {
public:
  int a;
  virtual void v();
};

class B : public virtual A {
public:
  int b;
  virtual void w();
};

class C : public virtual A {
public:
  int c;
  virtual void x();
};

class D : public B, public C {
public:
  int d;
  virtual void y();
};
```

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

vtableD

offset	value
+0	40 (vbase_offset)
+8	0 (top_offset)
+16	→typeinfo (D)
+24	→B::w()
+32	→D::y()
+40	24 (vbase_offset)
+48	-16 (top_offset)
+56	→typeinfo (D)
+64	→C::x()
+72	0 (vbase_offset)
+80	-40 (top_offset)
+88	→typeinfo (D)
+96	A::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

vtableD

offset	value
+0	40 (vbase_offset)
+8	0 (top_offset)
+16	→typeinfo (D)
+24	→B::w()
+32	→D::y()
+40	24 (vbase_offset)
+48	-16 (top_offset)
+56	→typeinfo (D)
+64	→C::x()
+72	0 (vbase_offset)
+80	-40 (top_offset)
+88	→typeinfo (D)
+96	A::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

vtableD

offset	value
+0	40 (vbase_offset)
+8	0 (top_offset)
+16	→typeinfo (D)
+24	→B::w()
+32	→D::y()
+40	24 (vbase_offset)
+48	-16 (top_offset)
+56	→typeinfo (D)
+64	→C::x()
+72	0 (vbase_offset)
+80	-40 (top_offset)
+88	→typeinfo (D)
+96	A::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

vtableD

offset	value
+0	40 (vbase_offset)
+8	0 (top_offset)
+16	→typeinfo (D)
+24	→B::w()
+32	→D::y()
+40	24 (vbase_offset)
+48	-16 (top_offset)
+56	→typeinfo (D)
+64	→C::x()
+72	0 (vbase_offset)
+80	-40 (top_offset)
+88	→typeinfo (D)
+96	A::v()

Wielokrotne dziedziczenie

D w pamięci

offset	value
+0	→ vtableD+24
+8	b
+16	→ vtableD+64
+24	c
+32	d
+40	→ vtableD+96
+48	a

vtableD

offset	value
+0	40 (vbase_offset)
+8	0 (top_offset)
+16	→`typeinfo (D)`
+24	→B::w()
+32	→D::y()
+40	24 (vbase_offset)
+48	-16 (top_offset)
+56	→`typeinfo (D)`
+64	→C::x()
+72	0 (vbase_offset)
+80	-40 (top_offset)
+88	→`typeinfo (D)`
+96	A::v()

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y

Więcej informacji

Itanium C++ ABI
http://www.codesourcery.com/public/cxx-abi/abi.html

(przystępne opracowanie)

Notes on Multiple Inheritance in GCC C++ Compiler v4.0.1
http://www.cse.wustl.edu/ mdeters/seminar/fall2005/mi.html

Uwaga!

Name mangling i implementacja funkcji/klas wirtualnych **zależą od kompilatora**.

Co jeszcze w C++

- konstruktory/destruktory obiektów (kolejność, wypełnianie `vtable`, alokacja pamięci)
- wyjątki
- template'y