Programowanie Niskopoziomowe

Notatki

„Mam nadzieję, że nie będzie na niskim poziomie.”
„To było na systemach operacyjnych.”

Maciej Mikołajczak
Kraków, 2025

Spis treści

1. Architektura x86 **2**

2. ABI Linuxa **3**

3. ABI C **5**

4. ABI C++ **6**

5. Mikroarchitektura CPU **7**

6. Instrukcje SIMD **9**

7. Self-Modifying Code **10**

8. Ataki niskopoziomowe **11**

9. Wielowątkowość **13**

10. Synchronizacja **14**

11. Obsługa I/O **16**

1. Architektura x86

2025-02-26

Procesor posiada pamięć wewnętrzną – rejestry. Są to pojedyncze komórki pamięci, są potrzebne, bo trzeba mieć jakąś minimalną pamięć potrzebną do komunikacji z RAM'em.

Pierwotnie były rejestry `ax, bx, cx, dx` i rejestry specjalne `si, di, sp, bp` – tak było w architekturze 16-bitowej.

Potem zmieniono na 32-bitowe – dodajemy do nazwy na początek `e` jak extended.

Na 64-bitach zamiast `e` jest `r` i dodano 8 nowych rejestrów – `r8, r9, ..., r15`.

Ostatecznie jest `rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, ..., r15`.

Stare nazwy dalej obowiązują – 32 dolne bity `rax` funkcjonuje pod nazwą `eax`, a dolne 16 – `ax`, do tego jest jeszcze `ah, al` – górna i dolna połowa `ax`, tak było od początku.

W rejestrach specjalnych wygląda to tak samo, nie ma podziału na części 8-bitowe, jest tylko dolna – `si -> sil`, nawet ona nie jest dostępna zawsze, tylko w niektórych instrukcjach.

W najnowszych rejestrach mamy nową nomenklaturę:

- `b` – jeden bajt
- `w` – słowo, dwa bajty (bo kiedyś była architektura 16-bitowa)
- `d` – double word, cztery bajty, bo to podwójne słowo gdy rozumiemy słowo jako 16 bajtów
- `q` – quad word, osiem bajtów

`r9 -> r9d, r9w, r9b` dają nam dostęp do dolnych części rejestru o danej długości.

Tryb pracy procesora może oznaczać ilość bitów używanych do adresowania pamięci lub ilość bitów rejestru. Standardowo myślimy o tym pierwszym. W procesorze 16-bitowym mamy maksymalnie 2^{16} adresów, to jest dość mało, dlatego wprowadzono mechanizm segmentacji – mamy 16 bitów segmentu i 16 offsetu, to znacząco zwiększa ilość dostępnej pamięci. Jako, że segment rzadko się zmienia, to można ustalić segment i dalej pracować na wskaźniku 16-bitowym, segment zmienić jak będzie to potrzebne. Dlatego będą do tego osobne instrukcje.

Początkowo pamięć nie była chroniona – operowano na adresach fizycznych, więc każdy mógł zmienić pamięć. Wzmocniono segmentację – system operacyjny ustala, gdzie programy widzą swoje segmenty i upewnia się, że nikt nie odwołuje się poza swoją pamięć. To się stało już, gdy były procesory 32-bitowe.

Drugi mechanizm – adresy wirtualne, adres ma normalnie 32 bity, dzielimy je na 12 dolnych i resztę – ta reszta oznacza stronę pamięci, która ma 2^{12} bitów. System operacyjny ustala to, do czego w pamięci fizycznej odnosi się strona. Dzięki temu mamy możliwość chronienia pamięci, jak i jej współdzielenia.

W ten sposób otrzymujemy następujące tryby pracy procesora:

- Legacy Mode:
  - Real Mode – 16 bitów, prosta segmentacja, nie chroniona pamięć, 1MB dostępnej pamięci.
  - Protected Mode – 32 bity, segmentacja, stronicowanie, chroniona pamięć, 4GB pamięci.
  - Virtual-8086 – uruchamianie 16-bitowych programów w trybie Protected.
- Long Mode:
  - 64-Bit Mode – brak segmentacji (bo już możemy adresować dużo pamięci), za to stronicowanie, chroniona pamięć, rejestry i adresowanie 64-bitowe.
  - Compatibility Mode – dla użytkownika działa jak Protected Mode, dla systemu jak 64-Bit.

W Legacy i Compatibility Mode mamy do dyspozycji kilka różnych segmentów, w 64-Bit Mode zostają nam tylko trzy – CS odpowiedzialny za przechowywanie atrybutów (np. uprawnień), FS i GS używane do robienia rzeczy thread-local.

W instrukcjach będziemy używać różnych sposobów adresowania:

- Absolute – pełny adres podawany w instrukcji.

- ModR/M:
$$\text{address} = \text{base} + \text{index} \cdot \text{scale} + \text{displacement},$$
gdzie base i index są w rejestrach, scale $\in \{0, 1, 2, 4, 8\}$ i displacement są podane w instrukcji. To pozwala nam na ładne przechodzenie po pamięci.
- Instruction-relative:
$$\text{address} = \text{RIP} + \text{displacement},$$
czyli przesuwamy się od obecnej instrukcji, to pozwala umieszczać dane wspólnie z kodem.
- Implicit – adres jest wyznaczony przez typ instrukcji.

W Assemblerze nie mamy typów zmiennych, tylko ciągi bitów. Ich interpretacja jest zawarta w instrukcji. Instrukcja identyfikuje rozmiar (byte, word, doubleword, quadword, double quadword), położenie (stała podana w instrukcji, rejestr, adres w pamięci, port I/O) i interpretację (signed int, unsigned int, BCD – Binary Coded Decimal, Packed BCD digits, strings, floats).

BCD trzyma każdą cyfrę w kolejnym bajcie, ale właściwie są do tego potrzebne 4 bity, dlatego Packed BCD trzyma po dwie cyfry w bajcie.

Pracujemy na wartościach wielobajtowych, można te bajty pisać od najmniej lub najbardziej znaczących. Jeśli wpiszemy „po ludzku" – od lewej do prawej, to najbardziej znaczący bajt ma najmniejszy adres. Dlatego drugi tryb też ma sens.

Little endian – pod najniższym adresem jest najniższy bajt, tak działa x86 i amd64.

Big endian – pod najniższym adresem jest najwyższy bajt, np. PlayStation i Xbox, protokoły sieciowe.

Wielka lista instrukcji:

- Transfer danych – `mov`, `cmovXX`, `push`, `pop`.
- Arytmetyczne – `add`, `sub`, `neg`, `mul`, `imul` (mnożenie naturalne i całkowite), `div`, `idiv`, `dec`, `inc`, `lea`.
- Logiczne i bitowe – `and, or, xor, not, shl, shr`.
- Testowanie i porównania – `cmp, test, bt, setXX`.
- Sterowanie przepływem – `jmp, jXX, loop, call, ret`.
- Wywołania systemowe – `syscall, sysret, sysenter, sysexit`.

Rejestr znaczników to rejestr, którego pojedyncze bity mają pewną interpretację. Niektóre np. mówią nam coś o wyniku ostatniej instrukcji. Jest np. carry flag – wynik wyszedł poza liczbę, nastąpiło przeniesienie, parity flag – w wyniku jest parzysta liczba jedynek.

Instrukcja `jXX` oznacza skok warunkowy, gdzie warunek zależy od któregoś ze znaczników. Podczas instrukcji `sub A, B` ustawiane są znaczniki, z których można np. wyciągnąć, która liczba jest większa (ważne jest to, że ustawią się różnie zależnie od tego, czy liczby mają znaki).

2. ABI Linuxa

2025-03-12

Podczas uruchamiania procesu najpierw sprawdzamy uprawnienia, potem alokujemy pamięć dla procesu (tworzymy przestrzeń adresową), kopiujemy sekcje inicjalizowane (np. kod do wykonania), ładujemy biblioteki linkowane dynamicznie, inicjalizujemy stos i rejestry, skaczemy na początek kodu (symbol `_start`). W C `_start` jest w bibliotece standardowej (tam jest inicjalizacja), która potem woła main.

Dane w przestrzeni adresowej są umieszczone w takiej kolejności (pierwsze punkty są na mniejszych adresach): kod i dane statyczne, sterta, biblioteki ładowane dynamicznie, stos.

Stos rośnie od góry do dołu a sterta od dołu do góry. Dlatego taka kolejność. O umieszczeniu bibliotek decyduje linker, przy starcie procesu daje się na sam dół, jak ładujemy coś w trakcie trwania procesu to idzie między stertę i stos.

Sterta istnieje dlatego, że stos jest zależny od wywołań funkcji. Pamięć, która ma przetrwać wywołania funkcji nie może na nim być.

W `/proc/<pid>/maps` mamy przestrzeń adresową (jej opis, w kolejności jak wyżej). Jeśli `<pid> -> self`, to widzimy swoją przestrzeń (procesu czytającego). Widzimy, że nad stosem jest coś jeszcze – to są jakieś informacje systemowe, o które proces może prosić. Są tam, żeby nie musiał robić wywołania systemowego.

Inicjalizacja stosu: na samej górze (czyli na najniższym adresie) `argc`, potem kolejne wskaźniki `argv[i]`, które wskazują na stos – dane są wpisane gdzieś wyżej. Potem jest 0 – widzimy, kiedy kończą się argumenty. Następnie wskaźnik na zmienne środowiskowe (napis z nimi). Potem 0, auxiliary entries, znowu 0 i na koniec rzeczy, na które wskaźniki były wcześniej.

Auxiliary entry to struktura, najpierw liczba oznaczająca rodzaj struktury, potem wartość (rozmiar stron, id użytkownika, jego grupa lub inne tego typu wartości).

Początkowy stan rejestrów:

- `rsp` – szczyt stosu.
- `rdx` – wskaźnik na funkcję do wywołania przed zakończeniem (raczej się nie używa).
- pozostałe – dowolnie.

Wywołania systemowe: `exit`, `open`, `read`, `write`, `brk`, `mmap`, `fork`, `exec`.

Robi się je za pomocą instrukcji `syscall` – nie ma argumentów, numer syscalla umieszczamy w `rax` (znajdujemy go w `/usr/include/asm/unistd_64.h`), kolejne argumenty w `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9` (konwencja wywoływania funkcji jest lekko inna – `rcx` zamiast `r10`). Nadpisywane są argumenty oraz `rcx` i `r11`, wynik w `rax` (od -4095 do 0, ujemny kod błędu to `errno`). W rejestrze można umieścić wskaźnik do pamięci (w naszej przestrzeni adresowej), w ten sposób przekazujemy systemowi większe informacje. Jeśli system ma dać nam więcej danych, to przekazujemy mu wskaźnik do miejsca, gdzie ma pisać. Przekazywanie tych danych odbywa się za pomocą struktur (pisane kolejno po sobie dane o odpowiednich rozmiarach).

Możemy w Assemblerze zapisać dane na koniec różnych sekcji (statyczna alokacja):

```
1  ; segment kodu
2  SECTION .text
3  ; d - data, b - byte, dane bajt po bajcie
4  db "halo"
5
6  ; dane inicjalizowane
7  SECTION .data
8  ; word - każda wartość to 2 bajty
9  dw 42, 54
10
11 ; dane nieinicjalizowane (to samo co wyżej, tylko bez zawartości
       początkowej)
12 SECTION .bss
13 ; nie można db, bo ma nie być zawartości, możemy zarejestrować daną ilość
       bajtów (lub innych jednostek, tutaj - 8 bajtów)
14 resq 4096
```

Dynamicznie możemy alokować pamięć:

- na stosie – przesuwamy wskaźnik na stos. Jeśli odwołamy się poniżej zaalokowanej pamięci stosu, to system nam go rozszerzy. Jest na to jakiś limit – nie możemy w ten sposób dostać bardzo dużo pamięci. Ta pamięć jest lokalna dla wywołania funkcji. Za to działanie ze stosem jest szybkie.
- syscall `brk` – działanie ze stertą, przesuwamy do góry adres końcowy sterty. Ten obszar jest zawsze spójny – jeśli zaalokujemy dwa bloki danych, to nie możemy zdealokować tego pierwszego bez zdealokowania drugiego. Dlatego proces musi sam zarządzać pamięcią (np. `malloc`).
- syscall `mmap` – mapuje plik do przestrzeni adresowej (w dowolne miejsce, jeśli nie podamy konkretnego adresu). Ten obszar może być współdzielony. Można też mapować strony pamięci bez wskazywania pliku (dostajemy pamięć). Możemy te strony niezależnie zwalniać (lepiej niż na stercie). Minus jest taki, że syscalle są wolne i zawsze dostajemy pełne strony (kwant po 4 KiB).

3. ABI C

2025-03-12

Pojawia się pojęcie wyrównywania (data alignment) – dokłada się sztuczne dane, aby adresy były „okrągłe". Pojedyncze zmienne nie muszą być wyrównane, ale warto to robić, bo jest to szybkie (procesor ma dostęp do pamięci za pomocą krótkich segmentów, jak nie ma wyrównania, to jedna zmienna może być pomiędzy segmentami). Wyjątek stanowią zmienne typu `packed` dla operacji SIMD (muszą być wyrównane zgodnie ze swoim rozmiarem).

W strukturach pola są wyrównane zgodnie ze swoim rozmiarem (`long double` do 16B), jeśli przed aktualnym polem występuje krótsze pole, to dodajemy wyrównanie do długości odpowiadającej aktualnemu polu. Do tego cała struktura jest wyrównana do tylu bajtów ile wynosi największe wyrównanie jej składowej.

Podczas wywoływania funkcji niektóre rejestry muszą być zachowane:

- `rbp`, `rbx`, `r12-r15` są callee-saved, funkcja ma obowiązek je przywrócić.
- pozostałe są caller-saved, funkcja może je zmienić.
- flaga `DF` (direction flag) powinna być wyzerowana na wyjściu i wejściu do funkcji, pozostałe można zmieniać.
- bity kontrolne rejestru `mxcsr` i `x87 control word` muszą zostać zachowane (są używane przy operacjach zmiennoprzecinkowych).

Argumenty funkcji są przekazywane w rejestrach:

- ogólnego przeznaczenia (GPR), czyli w `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.
- SSE, czyli 128-bitowe rejestry `xmm0` – `xmm7`.
- `al` – jest tam liczba argumentów przekazanych przez SSE.
- pozostałe są na stosie, wpisane od prawego do lewego (pierwszy na stos trafia najbardziej prawy argument).

Występują klasy argumentów, które mają przypisane to, w jakich rejestrach się znajdą:

- `INTEGER` – w rejestrach GPR.
- `SSE` – dolne 64 bity rejestru SSE.
- `SSEUP` – górne 64 bity rejestru SSE.
- `X87, X87UP` – przekazywane na stosie.
- `MEMORY` – również na stosie.

Liczby całkowite, `bool` i `void*` dostają `INTEGER`. Liczby zmiennoprzecinkowe i `__m64` dostają `SSE` (poza `long double`, to jest `X87, X87UP`). `__m128` i `__float128` dostają `SSE` i `SSEUP` (dzielą się na części), a `__m256` dostaje `SSE`, `SSEUP`, `SSEUP`, `SSEUP` (podział na 4 części).

Jeśli struktura ma ponad 16B lub jest wewnętrznie niewyrównana, to dostaje `MEMORY`. Inaczej elementy są klasyfikowane oddzielnie i łączone w grupy po 8B, dla których rozważamy przypadki: jeśli istnieje element `MEMORY`, całość dostaje `MEMORY`. Potem to samo, jeśli istnieje element `INTEGER`. Jeśli istnieje element `X87` lub `X87UP` całość dostaje `MEMORY`. W przeciwnym wypadku całość dostaje `SSE`. Jeśli któraś z grup jest `MEMORY`, to całość struktury dostaje `MEMORY`. Do tego w C++ klasy z nietrywialnym konstruktorem kopiującym lub destruktorem są przekazywane przez referencję jako `INTEGER`. Główny wniosek jest taki, że małe struktury mogą być przekazywane prze jeden rejestr – cała struktura może być `INTEGER` i zostać przekazana naraz.

Jeśli wynik funkcji jest klasy `MEMORY`, to musi ona otrzymać jako pierwszy argument adres miejsca na wynik, zwraca też ten adres w `rax`. `INTEGER` jest zwracany przez `rax`, a `SSE, SSEUP` przez `xmm0`, `X87, X87UP` przez `st0`. Struktura, która może być zwracana przez rejestry trafia do `rax`, `rdx`, `xmm0`, `xmm1`.

Przed wywołaniem funkcji stos powinien zostać wyrównany do 16B. Na stos wrzucany jest adres powrotu, potem argumenty.

Po wartości `rsp` jest „red zone” – tam są wartości, które być może są na stosie, ale `rsp` nie zostało zmienione, aby to odzwierciedlać (procesor wsadził wartości, ale nie przesunął jeszcze `rsp`, żeby potem zrobić duże przesunięcie na raz). Debuggery wiedzą, żeby nie zmieniać rzeczy w tym miejscu, mimo że teoretycznie nie ma tam stosu.

Statyczne łączenie modułów jest łatwe, bo linker dostaje te moduły i wsadza je w odpowiednie miejsca podczas linkowania. W Assemblerze nazwy funkcji są takie same jak w C (w C++ trzeba dodać do deklaracji `extern "C"`).

```
; import funkcji
EXTERN funkcja
call funkcja

; oraz zmiennej
EXTERN zmienna
mov rax, [zmienna]

; wprowadzenie symboli, aby były widoczne dla linkera
GLOBAL funkcja
GLOBAL zmienna
```

Biblioteki mogą być dołączane dynamicznie. Tak działają np. biblioteki współdzielone, bo nie chcemy, żeby każda binarka miała jej kopię w sobie. Wtedy segment kodu biblioteki może być współdzielony przez procesy (jest tylko jeden w RAM'ie), ale procesy widzą je pod różnymi adresami wirtualnymi. Do tego sekcje danych są oddzielne dla procesów korzystających z biblioteki. Zatem biblioteka powinna działać niezależnie od swojej pozycji (Position Independent Code), w szczególności nie mieć żadnego zahardcodowanego adresu w swoim kodzie.

Position Independent Code w trybie 64-bitowym wykorzystuje adresowanie względne. Wszystkie sekcje biblioteki są ładowane w spójnym bloku pamięci, odwoływanie się do danych wykorzystuje adresowanie względem RIP (`mov rax, [rel zmienna]`)

```
; przy dynamicznym eksportowaniu modułów podajemy typ - czy jest to zmienna,
    czy funkcja
GLOBAL symbol:function

; przy eksportowaniu danych możemy dodatkowo wskazać ich długość
GLOBAL tablica:data tablica.end-tablica
tablica: resd 128
.end:
```

Biblioteka współdzielona zostanie wsadzona w pewne nieznane miejsce przestrzeni adresowej. Dlatego w pewnym ustalonym miejscu istnieje Procedure Linkage Table (PLT) – zawiera wskaźniki do funkcji importowanych dynamicznie. Aby wywołać taką funkcję robimy `call printf wrt ..plt` – mówimy, że symbol jest w sekcji `..plt`. Pod takim symbolem znajduje się kod dynamic-linkera, który w razie potrzeby ładuje bibliotekę i uzupełnia referencje, a potem skacze do odpowiedniego kodu. Podobnie działają eksportowane dynamicznie dane. Mamy Global Offset Table (GOT), w której są adresy danych zewnętrznych.

```
EXTERN variable
mov rsi, _GLOBAL_OFFSET_TABLE_
mov rsi, [rsi + variable wrt ..got] ; znajdujemy adres i pod niego patrzymy
```

4. ABI C++

2025-03-26

W C nie może być dwóch funkcji o tej samej nazwie i różnych typach. W C++ mogą być. W Assembly muszą dostać różne nazwy. Dlatego występuje name mangling – dodajemy typ do nazwy funkcji. Jeśli chcemy tego uniknąć możemy użyć ABI języka C dodając `extern "C"` do deklaracji funkcji, ale wtedy nie możemy przeładowywać jej nazwy.

Aby zobaczyć symbole w pliku robimy `objdump -t file.o`. Tłumaczenie zmanglowanych nazw na coś czytelnego za pomocą `c++filt` – bierze na wejście jakiś tekst i demangluje wszystko, co wygląda na symbol.

Referencje to w kodzie maszynowym normalne wskaźniki. Metody obiektów to zwykłe funkcje, które jako pierwszy argument dostają wskaźnik na `this`. Enkapsulacja istnieje tylko na poziomie C++.

Dane w strukturach są układane w taki sposób, że najpierw pomijamy wszystkie funkcje i umieszczamy zmienne jak w C. Metody niewirtualne to funkcje niezależne od struktur, leżą osobno, ze strukturą łączy je tylko ich nazwa.

Przy metodach wirtualnych (mogących się zmieniać zależnie od typu dynamicznego obiektu) trzeba się bardziej wysilić. W językach dynamicznych typu Python metody dynamiczne są po prostu wskaźnikami w obiekcie. Wtedy jednak każda instancja musi mieć swój osobny wskaźnik do każdej metody – duże zużycie pamięci, a na pewno można lepiej, bo w C++ nie określamy tej metody na poziomie instancji, tylko klasy – znając dokładny typ obiektu znamy implementację wszystkich metod wirtualnych. Dlatego w instancji jest wskaźnik do tabeli metod wirtualnych (vtable), w niej są wskaźniki do implementacji metod. Taka tabela jest jedna na całą klasę.

Pierwszym wpisem w strukturze jest wskaźnik na vtable. To jest wskaźnik na opis pierwszej metody wirtualnej, czyli zazwyczaj `vtable+16` – pod `vtable` znajduje się `top_offset` (informacja o tym, jak daleko wskaźnik w strukturze wskazujący na vtable znajduje się od początku struktury), a pod `vtable+8` jest `typeinfo` – identyfikuje dokładny typ obiektu.

Jednokrotne dziedziczenie działa prosto – do struktury i vtable typu rodzica dokładamy na koniec trochę nowych rzeczy. Ciężej jest przy wielokrotnym dziedziczeniu. Mając `C : A, B` musimy tak ułożyć dane, aby dało się patrzeć na obiekt typu `C` jak na obiekt typu `B`. W tym celu w strukturze i vtable najpierw umieszczamy rzeczy pochodzące z `A`, potem w strukturze umieszczany jest drugi wskaźnik na vtable – tym razem na jej późniejszy fragment, w którym zaczynają się metody `B` (a nad nimi jest `top_offset` i `typeinfo`). Po nim następują dane obiektu `B`, a na końcu są dane obiektu `C` (tak jak przy jednokrotnym dziedziczeniu).

Przy przesłanianiu funkcji wirtualnych jest problem – jeśli nadpiszemy implementację w klasie-dziecku, to jako swój pierwszy argument będzie ona oczekiwać wskaźnika na obiekt tego typu, a nie typu rodzica, co jest problematyczne, gdy wykonujemy ją przez obiekt zcastowany na typ rodzica. Aby sobie z tym poradzić zastępujemy wskaźnik na taką funkcję w vtable przez tak zwany thunk – fragment kodu, który przesunie wskaźnik na obiekt do tego, którego oczekuje implementacja wywoływanej metody, a dopiero potem zawoła metodę. Thunki muszą być różne dla różnych funkcji, bo muszą mieć w sobie zhardcodowane wartości (a nie po prostu korzystać z `top_offset`) – metoda może być przesłonięta na innym poziomie niż najwyższy.

W przypadku wielokrotnego dziedziczenia po tym samym typie trzymamy kilka kopii podobiektów tego typu. Inaczej jest przy dziedziczeniu wirtualnym – wszystkie wirtualne dziedziczenia po danym typie są zaspokajane przez wsadzenie struktury tego typu na koniec obiektu. W szczególności można dziedziczyć po jakimś typie wirtualnie i niewirtualnie, wtedy niewirtualne dziedziczenie dostanie swój osobny obiekt tak jak zazwyczaj.

To, gdzie są podobiekty wirtualne zależy od struktury całej klasy (bo są wsadzane na koniec). Zatem informacja o tym musi być zawarta w vtable. Nad każdym `top_offset` umieszczamy dodatkowo `vbase_offset` dla każdej klasy wirtualnej (w kolejności preorder na listach dziedziczenia) – informacja o tym, o ile trzeba przesunąć wskaźnik, aby dostać obiekt danego typu.

Podobnie thunki nie mogą mieć w sobie stałej jak przedtem, więc do vtable dodajemy `vcall_offset` (o ile trzeba przesunąć wskaźnik na obiekt klasy wirtualnej, aby otrzymać wskaźnik na obiekt typu, który nadpisuje metodę wirtualną). Umieszczamy go nad `top_offset` klasy wirtualnej. Każda metoda wirtualna dostaje swój.

5. Mikroarchitektura CPU

2025-04-09

Kiedyś RAM był równie szybki jak CPU. W tej chwili procesory są dużo szybsze niż pamięć – zjawisko memory wall. Procesory są ograniczane przez dostęp do pamięci. Aby z tym walczyć można przesyłać więcej danych naraz. Komunikacja działa tak, że mamy szynę adresową, gdzie procesor daje adres, a RAM

zwraca dane na szynie danych. Im szersza szyna danych, tym więcej danych procesor może odczytać na raz.

Double Data Rate (DDR) – przy przesyłaniu danych wysyłamy dwie paczki danych. Jest to tańsze w realizacji niż poszerzenie szyny, bo nie wymaga zwiększenia ilości pinów w procesorze i pamięci. Jest też technologia Dual Channel – mamy dwie jednostki RAM'u, zapytanie o adres działa tak, że nie ma ustalonego ostatniego bitu. Jeden RAM odpowiada na ten parzysty, drugi na nieparzysty. Z punktu widzenia procesora wygląda to jak DDR. Można też łączyć te techniki.

Powyższe pomysły dają przyspieszenie, gdy odwołujemy się do sąsiadujących adresów. Programy jednak często odwołują się do różnych adresów, ale małej ich ilości. Aby to przyspieszyć umieszczamy pamięć pod tymi adresami „bliżej" – w pamięci podręcznej. Taka pamięć jest droga (bo musi być bardzo mała). Dlatego mamy kilka poziomów pamięci. RAM jest najwolniejszy, ale dość duży. Do tego jest kilka poziomów pamięci cache.

Pamięć cache ma parametry: rozmiar pamięci, rozmiar linii (jednostka, po której pamięć jest skwantowana), czas dostępu (latency – jak szybko odpowiada, liczone w cyklach procesora) oraz associativity – każdy adres ma przypisane miejsce w cache'u, do którego trafi (ustalone na podstawie kilku ostatnich bitów). Jeśli mamy dwa adresy trafiające w to samo miejsce, to będą się nawzajem wyrzucać. Tego typu architektura ma associativity 1. Ta wartość może być większa – każde miejsce w cache'u może trzymać kilka adresów, jest jakiś algorytm zarządzania tym, co trzymamy. Duże associativity zwiększa latency oraz powoduje, że jednostka jest droższa (bo musi więcej robić). Dlatego pamięć podręczna blisko procesora ma małe associativity, a pozostała trochę większe.

Jest kilka faz wykonywania instrukcji procesora: fetch (ściągnięcie danych), decode (rozpoznanie typu instrukcji, argumentów), prepare (przygotowanie argumentów), execute (wykonanie), store/commit/retirement (zapisanie wyniku, znaczników). Za te fazy odpowiadają różne części procesora. Gdyby fazy były wykonywane sekwencyjnie, to każda część przez większość czasu byłaby bezczynna. Dlatego fazy dla różnych instrukcji dzieją się równolegle na tyle, na ile mogą. Zależności między instrukcjami powodują, że czasem muszą na siebie czekać, co spowalnia procesor.

Rodzaje zależności między instrukcjami: kolejność (program jest sekwencyjny), zależność danych (wynik pierwszej instrukcji jest potrzebny do drugiej), zależność rejestrów (instrukcje zajmują te same rejestry), zależność przepływu (od wyniku instrukcji zależy, co będziemy robić dalej).

Można sobie radzić z tymi zależnościami:

- Kolejność – out-of-order execution (kolejne fazy nie są sekwencyjne, tylko faza commit musi być w odpowiedniej kolejności).
- Zależność danych – nie da się uniknąć.
- Zależność rejestrów – unikamy poprzez register renaming (rejestrów w procesorze jest więcej niż nazwanych rejestrów w architekturze, procesor ja mapuje, może być kilka rejestrów fizycznych trzymających dany rejestr), który działa wszędzie poza fazą commit.

Zależności przepływu mają bardziej skomplikowany system. Działa tam mechanizm speculative execution – przewidywanie skoku i wykonywanie instrukcji aż do fazy commit, która wymaga potwierdzenia skoku. Jeśli przewidzimy dobrze, to te zależności praktycznie znikają. Jeśli się pomylimy, to mamy znaczne opóźnienie, bo trzeba cofnąć rzeczy, które zrobiliśmy.

Do przewidywania skoku lub jego braku służy mechanizm branch prediction – mamy automat stanowy (często 2-bitowy saturating counter, czyli mamy 4 stany od „strongly jump" do „strongly not jump" i każdy skok lub jego brak zmienia stan o jeden w odpowiednią stronę). Takich liczników jest wiele, każdy jest przypisany do danej instrukcji (do jej adresu fizycznego). To dobrze przewiduje długie pętle (dużo razy skok, potem nie), ale bardzo źle się zachowuje na ciągłych zmianach.

Dlatego pamiętamy też historię skoków. Mamy tablicę rozmiaru 2^k, indeks w tablicy to historia ostatnich k skoków (maska bitowa), podejmujemy decyzję na podstawie wartości w tablicy. To jednak wymaga dużo więcej miejsca, więc nie możemy mieć tak dużo lokalnych liczników. Dlatego są rozwiązania hybrydowe, np. nie mamy licznika na każdą instrukcję, tylko licznik jest przypisywany do instrukcji na podstawie jej adresu i kilka instrukcji może mieć ten sam licznik.

Z tego wszystkiego wynika, że zależności danych i przepływu sterowania powodują opóźnienia rzędu kilkunastu cykli, a dostęp do pamięci rzędu kilkuset cykli. Aby optymalizować dostępy do pamięci należy

umieszczać używane razem funkcje i dane obok siebie, dbać o alignment i unikać alokacji dynamicznej (alokacja jest kosztowna i ma nieoptymalny alignment) oraz organizować dane tak, aby jak najczęściej były odczytywane sekwencyjnie.

Problematyczne są skoki warunkowe, w których rozkład skoków jest równomierny lub gdy jest ich dużo pod rząd. Podobnie jest z wywołaniami funkcji. Aby temu zapobiegać stosuje się inlining funkcji oraz unika skoków warunkowych, np. przez instrukcje `SETcc` lub `CMOVcc`.

6. Instrukcje SIMD

2025-04-16

Instrukcje SIMD (Single Instruction, Multiple Data) służą do wykonania tej samej operacji na dużym zestawie danych. Wykorzystują one osiem 64-bitowych rejestrów `mmx0-mmx7` (współdzielonych z FPU) oraz szesnaście 128-bitowych rejestrów `xmm0-xmm15`, które są rozszerzane do 256-bitowych `ymm0-ymm15`. Możemy działać na liczbach zmiennoprzecinkowych o 32 lub 64 bitach albo na liczbach całkowitych o 8, 16, 32 lub 64 bitach (ze znakiem lub bez).

Instrukcje SIMD mogą kończyć się z błędem, np. przy utracie precyzji, dzieleniu przez zero, otrzymaniu zdenormalizowanego argumentu. Wykonujemy kilka operacji na raz, mogą zakończyć się z różnym statusem. Wystąpienie błędu jest rejestrowane poprzez ustawienie odpowiedniej flagi w rejestrze `mxcsr` (pierwsze kilka bitów). Pozostałe bity odpowiadają za maskowanie – można je zapalić, co spowoduje, że procesor nie będzie generował wyjątków przy wystąpieniu odpowiednich błędów.

Mamy kilka grup instrukcji, są to między innymi instrukcje służące do przesyłania danych, wykonywania operacji zmiennoprzecinkowych lub całkowitoliczbowych, konwersji typów i unikania sterowania przepływem.

Do przesyłania danych mamy różne instrukcje zależnie od tego, co i gdzie przesyłamy:

- `movd` przesyła liczbę o 32 lub 64 bitach między rejestrem xmm a zwykłym rejestrem (gpr) lub pamięcią.
- `movq` przesyła 64 bity w ten sam sposób i zeruje pozostałe (górne) bity rejestru xmm (zero-extend).
- `movdqa` przysyła 128 bitów (double quadword) danych wyrównanych do 16 bajtów między xmm a xmm lub pamięcią. Wykonanie tej operacji na niewyrównanych danych powoduje błędy. Bezpieczniejsza (ale wolniejsza) jest `movdqu`.
- `movdq2q` przysyła xmm do mmx (double quadword to quad), na odwrót działa `movq2dq` (ponownie mamy zero-extend).
- `movntps, movntdq, movntpd` i inne instrukcje postaci `movnt??` to instrukcje non-temporal, przy których dane nie trafiają do cache'u, co może się przydać, gdy chcemy skorzystać z danych tylko raz.
- `prefetch0, prefetch1, prefetchnta` sugerują procesorowi, by preloadował jakieś dane (mniejsze liczby oznaczają lepszy cache, do którego mogą trafić dane). Są to sugestie dla procesora, niekoniecznie się wykonają.

Mamy instrukcje pakujące, np. `packssdw`, która w pierwszym argumencie dostaje rejestr xmm zawierający 4 liczby 32-bitowe, a w drugim rejestr xmm lub pamięć zawierającą inne liczby 32-bitowe. Te liczby są obcinane do 16 bitów (po wartościach – duże na moduł liczby przyjmują maksymalną reprezentowalną wartość) i umieszczane w pierwszym rejestrze. Podobnie działa `packsswb` (16 bitów na 8) i `packuswb` (to samo, tylko bez znaku).

Podobnie możemy rozpakowywać. `punpckhbw` dostaje rejestr xmm i drugi argument będący xmm lub pamięcią. Bierzemy górną połowę pierwszych i drugich danych i na zmianę (po 8 bajtów) wsadzamy je do wyniku. Wynik jest pisany do pierwszego rejestru. W szczególności jeśli drugi argument jest samymi zerami, to górne liczby 8-bitowe pierwszego rejestru zostaną rozszerzone do liczb 16-bitowych. Podobnie działa `punpcklbw`, tylko dla dolnej połowy. Z kolei `punpckhwd` robi to samo dla paczek po 16 bitów.

Możemy też mieszać dane. `pshufd` bierze xmm (destination) i xmm lub pamięć (source), trzeci argument to 8-bitowa liczba, jej pierwsze dwa bity oznaczają na którą z czterech pozycji w destination wsadzić pierwsze 32 bity source'a. Kolejne dwa oznaczają pozycję kolejnych 32 bitów i tak dalej. `pshufhw` robi to samo dla 16-bitowych paczek z górnej połowy rejestru, a `pshuflw` z dolnej.

Operacje całkowitoliczbowe możemy wykonywać z przepełnieniem (ignorowanie przepełnienia), np. `paddb` (16 liczb 8-bitowych), `paddw` (liczby 16-bitowe), podobnie `psub?` albo `pmul?`. Możemy też wykonywać je z wysyceniem (mamy największą i najmniejszą liczbę, których nie da się przekroczyć), np. `paddsb` (liczby 8-bitowe ze znakiem), `padduw` (16-bitowe bez znaku) i inne podobne. Mamy też operacje logiczne `por, pand, pandn, pxor`.

Na liczbach zmiennoprzecinkowych mamy operacje skalarne (pojedyncze liczby), np. `addss` (32-bity), `addsd` (64-bity). Mamy też operacje wektorowe „pionowe" (operacja na odpowiadających sobie liczbach z różnych zestawów danych) `addps` (4 liczby po 32-bity), `addsubps` (dodajemy drugą i czwartą, odejmujemy pierwszą i trzecią) i „poziome" `haddps`, `hsubpd` (operacja na sąsiednich liczbach z tego samego zestawu danych).

Możemy też wykonywać bardziej złożone operacje, np. `sqrtps` (pierwiastek 32-bitowy), `rcpps` (odwrotność) lub konwersje instrukcjami `cvt??2??`, gdzie za pytajniki można wstawić `ss` (32-bity w xmm), `ps` (4 razy 32-bity w xmm), `si` (32 lub 64 bity w gpr) i inne podobne. Możemy brać maksimum i minimum instrukcjami typu `pmaxub`. Porównania wykonujemy za pomocą `comiss`, `comisd` (pojedyncze liczby zmiennoprzecinkowe), `pcmpeqb` (równość 16 liczb 8-bitowych) i innych podobnych.

7. Self-Modifying Code

2025-04-23

SMC (Self-Modifying Code) to kod, który generuje lub modyfikuje instrukcje w pamięci w trakcie wykonywania. Coś takiego dzieje się podczas kompilacji Just-In-Time (JIT). Służy też do optymalizacji (kod zmienia się zależnie od architektury komputera, by dostosować się do niej). SMC pozwala też utrudnić zrozumienie kodu (co robi np. malware).

SMC można robić na kilka sposobów. Pierwszy to patching – modyfikacja istniejącego kodu w miejscu. Trzeba mieć uprawnienia do pisania po segmencie kodu, istnieje co najwyżej jedna wersja kodu naraz i zmiana może wyczyścić pipeline. Ta metoda jest więc najprostsza, ale ma sporo minusów.

Kopiowanie i patching – kopiujemy kod i zmieniamy w nim stałe (lub coś więcej). Wymaga to alokacji pamięci z uprawnieniami `rwx`. Jeśli robimy coś więcej niż zmiana stałych, to musimy zachować rozmiar poszczególnych instrukcji w kopiowanym bloku i poprawić adresy względne w różnych blokach (np. występujące w pętlach). Można też generować od zera – alokacja pamięci z uprawnieniami `rwx`, do tego jest czasochłonne i skomplikowane.

Do zmiany uprawnień do stron pamięci służy syscall `mprotect`.

W językach interpretowanych mamy interpreter, który patrzy na kolejne linie i jest wielkim switchem (słaba skuteczność branch prediction). Do tego mamy dwa zestawy kodu i danych (programu i interpretera), co powoduje słabą lokalność referencji. Z tych powodów języki interpretowane są wolne. Rozwiązaniem jest kompilowanie kluczowych fragmentów kodu.

Leniwe ładowanie i kompilacja kodu (funkcji) działa tak, że zamiast funkcji jest thunk zawierający kod loadera lub kompilatora, po pierwszym wywołaniu funkcja jest ładowana (kompilowana) i zastępuje thunk, kolejne wywołania używają już istniejącej funkcji.

Format instrukcji w x86-64 (kolejne fragmenty położone pod rosnącymi adresami, cała instrukcja ma co najwyżej 15 bajtów):

- Legacy Prefix, może być kilka takich, każdy ma 1 bajt.
- REX Prefix, który też można pominąć, ma 1 bajt.
- Opcode, czyli rodzaj instrukcji, 1 lub 2 bajty.
- ModRM, czyli sposób adresowania pamięci, 1 bajt.
- SIB, czyli dalsza część sposobu adresowania, 1 bajt.
- Displacement, czyli offset wykorzystywany przez ModRM i SIB, zajmuje od 1 do 8 bajtów.
- Immediate memory (jakaś stała), też do 8 bajtów.

Możliwe prefiksy to operand-size override (zmienia domyślny rozmiar operandu, czyli 32 bity na 16), address-size prefix (zmienia domyślny rozmiar adresu, czyli 64 bity na 32), segment override (decyduje,

którego segmentu instrukcja ma używać), lock (wymusza atomowość niektórych operacji), repeat (prefiksy `REP, REPE, REPNE`).

Prefix REX pojawił się, bo w trybie 64-bitowym mamy więcej rejestrów i kodowanie ich identyfikatorów zajmuje więcej bitów. Górne 4 bity REX są ustalone (0100), potem jest `REX.W` – to, czy rozmiar operanda ma 64 bity. Potem `REX.R, REX.X, REX.B` – najwyższy bit identyfikatora rejestru, który jest odpowiednio w `reg` w ModRM, w `index` w SIB oraz w `r/m` w ModRM lub w `base` w SIB (tylko jeden z tych dwóch może występować w jednym momencie).

Jeśli opcode zajmuje dwa bajty, to pierwszy ma wartość `0f` (kiedyś były jednobajtowe opcode'y, potem potrzebne było więcej – ta wartość oznacza, że patrzymy na drugi zestaw opcode'ów). Poza instrukcją opcode może kodować jeden z argumentów (będący rejestrem). Każda instrukcja może adresować pamięć co najwyżej raz i jeśli operand, który może być adresem, może być na kilku pozycjach, to każda z tych opcji na swój własny opcode.

ModRM (mode register memory) mówi o tym, w jaki sposób instrukcja adresuje pamięć. Dzieli się na `mod`, `reg` i `r/m`, które mają odpowiednio 2, 3 i 3 bity. `mod` i `r/m` określają tryb adresowania operandu, który może być adresem (jest oznaczony `reg/mem`). `reg` określa rejestr drugiego operandu lub (gdy go nie ma) stanowi rozszerzenie opcode'u. Dla większości wartości `mod` w `r/m` jest adres bazowy, do którego jest dodawany displacement (który ma różną długość zależnie od wartości `mod`). Dla `mod = 11` w `r/m` jest rejestr. Dla `mod = 00` i `r/m` odpowiadającego `rbp` (101) adresujemy względem `rip`. Dla dowolnego `mod` i `r/m` odpowiadającego `rsp` (100) używamy SIB.

Podobnie SIB dzieli się na `scale`, `index` i `base`. Zależnie od `mod` i tych wartości adresujemy relatywnie na różne sposoby. Są one postaci `base+scale*index+displacement` w różnych konfiguracjach (różne displacementy, brak skalowania).

8. Ataki niskopoziomowe

2025-04-30

Ataki typu buffer-overrun wymagają wczytywania danych nieznanego rozmiaru ze źródła kontrolowanego przez napastnika. Jeśli programista nie zaimplementował kontroli rozmiaru, to możliwe są ataki stack-based i heap-based buffer-overrun. Opiszemy ten pierwszy.

Na stosie mamy adres powrotu, poprzednią wartość `rbp` i zmienne lokalne, wśród których jest nasz bufor. Możemy dostać tak dużo danych, że nadpiszą adres powrotu – skok pod adres, który podał napastnik. W szczególności może skoczyć do wnętrza bufora i wykonać z niego kod. Taki atak może mieć miejsce, jeśli napastnik faktycznie może przesłać dane aż do adresu powrotu, strony pamięci zawierające stos mają prawa do wykonywania, a do tego napastnik zna bezwzględny adres bufora na stosie.

Bardziej wyrafinowany scenariusz: napastnik może dostarczyć jeden dodatkowy bajt (czasem o ustalonej wartości, np. przy wczytywaniu stringa to musi być 0). Jeśli bufor jest pierwszą zmienną lokalną, to napastnik jest w stanie wyzerować najmniej znaczący bajt `rbp`. Poprzednia funkcja spodziewa się, że w `rbp` będzie adres początku jej ramki i do niego skacze pod koniec wykonania. Jeśli poprzednia funkcja jest krótka, to przy zakończeniu może skoczyć do bufora.

Hardware'owo można się przed tym zabezpieczyć ustawiając bit `NX` – uniemożliwia interpretowanie stosu jako instrukcji. Zazwyczaj nie ma stron pamięci, które mają jednocześnie prawa do zapisu i wykonywania. Zatem napastnik musi wykorzystać już istniejące bajty, np. z kodu programu albo wykorzystywanych bibliotek.

Tego typu scenariusz nazywa się Return-Oriented Programming (ROP). Napastnik może skoczyć do dowolnego adresu w programie. Wykona się kilka instrukcji, po których nastąpi `ret` – znów do miejsca wskazanego przez napastnika. W ten sposób jesteśmy w stanie wykonać cały ciąg instrukcji. Stos poza adresami może zawierać też dane (wybieramy kod z instrukcją `pop`), a więc możemy wstrzykiwać dane. Można też sterować przepływem wykorzystując `jmp` i `call` z warunkiem zawierającym rejestr. Istnieją narzędzia, które wyszukują odpowiednie fragmenty w kodzie i podają je napastnikowi. Istnieją nawet kompilatory, które korzystają z dostępnych instrukcji, a napastnik może po prostu pisać zwykły kod.

Systemowe zabezpieczenie ASLR (address space layout randomization) układa pod losowymi adresami: adresy bazowe bibliotek współdzielonych, stos i stertę (dane dynamiczne) oraz kod i dane statyczne programu kompilowanego jako position-independent code. Teraz napastnik nie zna bezwzględnych adresów, a więc ciężej mu podać odpowiednie adresy powrotu.

W programach nie-PIC możliwe jest ominięcie ASLR. Na pewno można korzystać z kodu samego programu, ale raczej nie ma w nim instrukcji `syscall` (bo to zwykle robi biblioteka standardowa, a potrzebujemy ich do eskalacji uprawnień). Wywołania funkcji z biblioteki korzystają z PLT, ich adresy są w GOT i są wypełniane przez linker dynamiczny. Adres GOT nie jest randomizowany, a randomizacja nie zmienia względnego położenia instrukcji w bibliotekach. Zatem można zmienić wartość w GOT o właściwy offset – możemy wykorzystać cokolwiek z biblioteki.

Zabezpieczamy się dalej – tym razem z poziomu kompilatora. W GCC mamy flagę `-fstack-protector`. Pomiędzy adres powrotu a zmienne lokalne wsadzana jest pseudolosowa wartość („canary") zmieniana co uruchomienie programu. Przed instrukcją `ret` sprawdzana jest zgodność tej wartości z oryginalną. Jeśli wartość jest inna, to program kończy wykonanie. W programach serwerowych korzystających z `fork` można ominąć ten mechanizm. Wartość canary w różnych podprocesach jest taka sama. Wysyłamy zapytanie nadpisujące jeden bajt, jeśli się zgodzi, to wszystko zadziała, w przeciwnym wypadku dostaniemy błąd. Możemy tak zgadnąć wartość canary, bo ta wartość się nie zmienia (wywaliliśmy podproces, ale rodzic dalej działa). Jeden bajt zgadujemy na pomocą 256 prób, a więc po około dwóch tysiącach zapytań mamy wartość canary.

Podobnie możemy ominąć ASLR, który losuje adres bazowy, ale utrzymuje offsety. Zatem wystarczy zgadnąć jeden adres, by mieć wszystkie. Mając jakiś adres powrotu w kodzie zmieniamy go po bajcie, jeśli zmienimy go na złą wartość, bo program się wywali. Jeśli nie (nadpiszemy tym samym, co było), to znamy jego wartość. W ten sposób znajdujemy całość. W większości systemów mamy stały prefiks i sufiks adresu, część losowa ma tylko 28 bitów, zatem brut-force po jednym bajcie zadziała po mniej niż tysiącu zapytań.

W przypadku ataku na stercie nie możemy nadpisać adresu powrotu, ale możemy nadpisać struktury danych programu, na przykład wskaźnik do funkcji lub tablicy metod wirtualnych. Wtedy program przy wykonaniu tej funkcji skoczy do wskazanego przez nas adresu. W ten sposób jesteśmy w stanie rozpocząć proces skakania po kodzie jak w ROP.

Mamy jeszcze ataki Spectre i Meltdown, które wykorzystują cache i zjawisko memory wall. Spectre oparty jest o speculative execution, pozwala odczytać dowolną wartość dostępną dla procesu. Wykorzystuje istnienie kodu, który wyglada tak:

```
if (index < size) {
    a = data[index];
    b = a&1;
    c = data[15+b];
}
```

Czytamy coś z tablicy (sprawdzając poprawność indeksu), a następnie wykorzystujemy tę wartość do wyliczenia kolejnego indeksu. Najpierw wyrzucamy z cache'u zmienną występującą przy sprawdzaniu poprawności indeksu (`index` lub `size`) oraz możliwe wartości danych, pod które ostatecznie chcemy sięgnąć (tutaj `data[15]` i `data[16]`). Wiemy, w jakich kubełkach cache'u będą te wartości, a więc możemy zmusić procesor, żeby wstawił tam jakieś inne dane.

Wykonujemy ten kod dla odpowiednio dobranego `index`, nawet dla niespełnionego warunku speculative execution zacznie działać, być może dojdzie do linijki `data[15+b]` i wczyta tę wartość do cache'u. Wtedy porównujemy czas dostępu do `data[15]` i `data[16]`, na podstawie tego wiemy, która wartość została sczytana, a więc wiemy, co było pod `a` (w tym wypadku tylko ostatni bit). W ten sposób możemy poznać dowolne wartości w pamięci, na przykład jakieś klucze kryptograficzne. Ten atak uderzył zwłaszcza w Javascript i przeglądarki. Wprowadzono różne zabezpieczenia, w szczególności usunięto możliwość dokładnego pomiaru czasu.

Meltdown jest oparty o out-of-order execution odczytu pamięci i kontroli uprawnień. Podobnie jak w Spectre odczytujemy niedostępne dla nas dane i patrzymy pod adres zależny od nich. Na podstawie czasu dostępu dostajemy informację o tym, co było w tych niedostępnych danych. Nie wymaga to wykonywania kodu przez atakowany proces. Pozwala odczytać dowolną wartość z przestrzeni adresowej atakującego, nawet jeśli nie ma on do niej uprawnień. W szczególności pamięć jądra miała kiedyś ten sam adres w każdej przestrzeni adresowej, by optymalizować context-switching. Zatem Meltdown pozwalał na czytanie danych jądra, w których są wszystkie wrażliwe dane.

9. Wielowątkowość

2025-05-14

Zamiast SIMD jest MIMD (Multiple Instructions, Multiple Data). Działa task switching – mamy jeden procesor i pamięć, możemy wykonywać jedno zadanie naraz, ale przełączamy się pomiędzy zadaniami. Istnieją też systemy współbieżne – jest wiele procesorów z wieloma rdzeniami, które współdzielą pamięć. Często działa hyperthreading – zwiększamy ilość jednostek, które są odpowiedzialne za dekodowanie instrukcji i podłączamy je do tych samych jednostek wykonawczych i cache'u, żeby zasypać je obliczeniami z różnych procesów. Kolejnym krokiem są systemy rozproszone – osobne pamięci RAM i systemy, które czasem są synchronizowane. Takie systemy muszą się komunikować.

Modele komunikacji w MIMD to: shared-everything (wspólna pamięć, przestrzeń adresowa i inne zasoby), shared-nothing (zasoby prywatne, trzeba przesyłać je w razie potrzeby) oraz modele hybrydowe (można wybrać część rzeczy do współdzielenia). Sposoby schedulingu zadań to cooperative (procesy same decydują o przekazaniu procesora) oraz pre-emptive (zasoby przydziela i odbiera system).

Procesy działają z komunikacją shared-nothing (poza współdzielonymi fragmentami) i pre-emptive scheduling. Zarządzanie ich zasobami jest proste (działa system), a synchronizacja i komunikacja jawna (nie mają wspólnych danych). Wymagają za to dużo odwołań do systemu i narzut na komunikację jest spory. Wątki systemowe (threads) są shared-everything (lub hybrydowo) i jest pre-emptive scheduling. Jest łatwo o komunikację (wspólne zasoby), a zarządzanie zasobami jest podobnie proste jak przy procesach. Dalej wymaga to odwołań do systemu (przy zmianie wątku), a do tego łatwo się zabugować przy współdzielonej pamięci. Wątki użytkownika (fibers) działają z shared-everything (lub hybrydowo) i z cooperative scheduling. Mamy mały narzut na komunikację i brak odwołań do systemu, ale za to trzeba zarządzać zasobami samemu i synchronizować dane.

W POSIXowych wątkach wspólne (dla całego procesu) są identyfikatory PID, UID, GID, zmienne środowiskowe, obecne directory, sterta, file descriptory, handlery sygnałów, biblioteki współdzielone i wszystko związane z IPC (pipe'y, semafory). Każdy wątek ma swoje ID, wskaźnik na stos (standardowo stos wątku nie rośnie tak jak normalnego procesu, jest ograniczony), rejestry, maski sygnałów, errno i inne dane dotyczące tylko wątku oraz priorytety w schedulingu. W Linuxie wątki mają swój własny PID.

W C jest biblioteka `pthread.h`, jest typ `pthread_t`, który w Linuxie jest po prostu longiem, ale teoretycznie może być czymkolwiek.

Wątek może się zakończyć poprzez return, wyjście (`pthread_exit`) i kooperatywne wyjście (funkcje `pthread_cancel` i `pthread_testcancel`), czyli pojawiają się prośby o zakończenie od innych wątków, na które można odpowiedzieć – wykonujemy funkcję, która nas zakończy, jeśli ktoś o to poprosił. Ciężko jest to obsłużyć, bo właściwie nie wiemy, czy nasz wątek się skończy. Trochę pomagają w tym funkcje `pthread_cleanup_push` i `pthread_cleanup_pop` – jedna dostaje funkcję (być może makro) i odkłada na stos, a druga ściąga ją i wykonuje lub nie, zależnie od przekazanej flagi. Można czekać na zakończenie innego wątku (`pthread_join`) lub odłączyć się (`pthread_detach`).

Konteksty wykonania muszą się komunikować i synchronizować. Mamy przede wszystkim mutexy i condition variables (warunki), które działają z pamięcią współdzieloną. Do tego mamy semafory, sygnały, wiadomości (IPC) i zdarzenia (events).

Mutex to zmienna, którą można zablokować (zajmując zasób) lub odblokować. Można robić dziwne rzeczy – zamykać zamknięty mutex, odblokować nie swój mutex i odblokować niezablokowany. Zwykły mutex (`NORMAL`) nie obsługuje tych sytuacji – w pierwszej następuje deadlock, a w pozostałych może stać się cokolwiek. Mutex `ERRORCHECK` w takich sytuacjach zwraca błąd. Mutex `RECURSIVE` może blokować się wielokrotnie. Mutexy zazwyczaj służą do ochrony przez równoległym pisaniem lub odczytem na współdzielonych danych. Mutex blokuje dane niezależnie od tego, co z nimi robimy. Istnieje reader-writer lock – blokujemy się na pisaniu lub czytaniu, na czytaniu może blokować się wiele wątków, ale na pisaniu tylko jeden.

Podstawowe zastosowanie mutexów to zabezpieczenie jakiejś sekcji danych przed niespójnością. Poza tym wątki czasem muszą się zsynchronizować, by dobrze współpracowały ze sobą. Zaimplementowanie tego mutexami jest karkołomne, bo jako wątek możemy poczekać (zablokować się) na mutexie tylko, jeśli ktoś wcześniej go zablokował, a ciężko jest mieć pewność, który wątek zacznie działać pierwszy – całkiem prawdopodobne, że będzie to ten, który powinien poczekać.

Będziemy chcieli móc blokować jakieś zasoby (dane), czekać na sygnał od innego wątku i przełączać się

pomiędzy tymi trybami atomowo – często chcemy, aby jeden wątek zablokował dane, zrobił coś z nimi, odblokował i dał sygnał, że są gotowe, a drugi poczekał na dane, zablokował je, przetworzył i odblokował. Brak atomowości tworzy race condition przy przełączaniu się pomiędzy trybami czekania. Do tego właśnie służą condition variables, które implementują dokładnie to przejście. Mamy funkcję `pthread_cond_wait`, która czeka na danej condition variable i z podanym mutexem. Funkcje `pthread_cond_broadcast` i `pthread_cond_signal` wysyłają sygnał do zadanego condition variable, pierwsza funkcja wysyła go do wszystkich czekających wątków, druga stara się do jednego. Występuje *spurious wakeup* – signal może obudzić kilka wątków, a wątek może się obudzić nawet bez sygnału.

Przy wątkach trzeba uważać na zarządzanie stosem, funkcje thread-safe (są funkcje zależące od globalnych danych, trzeba je synchronizować), thread-specific data i obsługę sygnałów. Problemy mogą się pojawić przy forkowaniu z wątku – kopiujemy tylko ten wątek, więc jeśli np. jakiś inny wątek trzyma mutex w momencie forka, to w procesie dziecku ten proces nie będzie istniał. Dlatego po forku najlepiej robić tylko exec. Trzeba też uważać na I/O, bo descriptory są wspólne.

W C++ mamy typ `thread`, któremu podajemy w konstruktorze funkcję do wykonania i jej argumenty. Można robić `join` i `detach` na wątkach. Jest typ `mutex`, ale raczej się go nie używa. Jest `lock_guard` – opakowanie na mutexa, które zamyka go w konstruktorze i otwiera w dekonstruktorze. Jeśli chcemy zamknąć wiele mutexów naraz możemy skorzystać z funkcji `lock` – zamknie je w deterministycznej kolejności, więc nie spowoduje to deadlocków. Po takim zamknięciu można przenieść mutexy do `lock_guarda`, żeby nie musieć ich potem zwalniać. Mamy też typ `condition_variable`, który może czekać (`wait`) na `unique_lock` skonstruowany z mutexa.

10. Synchronizacja

2025-05-21

Mamy sporo pułapek, na które trzeba uważać przy pisaniu wielowątkowego kodu. Kompilator może optymalizować kod, np. `while(!stop) {...}` może raz zapisać zmienną `stop` do rejestru i dalej patrzeć do niego. Rozwiązaniem jest deklaracja zmiennej jako `volatile` – kompilator nie może założyć, że wartość się nie zmienia. Ze względu na out-of-order execution może się zdarzyć, że instrukcje w sekwencyjnym kodzie wykonają się w odwrotnej kolejności. Normalnie nie zmienia to semantyki programu, ale działanie innych wątków może spowodować, że zajdą sytuacje, które nie mogłyby się zdarzyć w innej sytuacji.

Wykonywane operacje nie są atomowe (nawet pojedyncze instrukcje assemblera). Zatem robienie `x++` w różnych wątkach może przepleść inkrementacje i w rezultacie nie wykonać którejś. W Assembly do zapobiegania temu służy prefix `lock`, który znaczy, że instrukcja wykona się atomowo.

Mamy różne optymalizacje dostępu do pamięci: speculative reads (czytamy to, co może się przydać, czasem zgadując adres, pod którym będzie odczyt), buffered writes (zapis do cache'u zawsze odbywa się całymi liniami cache'u, więc czekamy z pisaniem, bo może przyjść więcej operacji na ten sam obszar), write combining (możemy nadpisać jedną pamięć kilka razy, buforowanie spowoduje, że wykona się tylko jeden zapis). Zatem odczyty mogą być wcześniej niż są zlecone, a zapisy później. Do tego każdy rdzeń ma swoją pamięć cache, w różnych cache'ach mogą być różne wersje tych samych danych. Cache coherency protocols zapewniają, że zapisane dane są synchronizowane. Wykonanie takiej komunikacji jest drogie, więc w Assembly trzeba ją wywołać explicite – mamy instrukcje ładowania do cache'u (`prefetch`) i pisania z niego do pamięci (`clflush`).

Do wymuszania kolejności operacji nowe kompilatory mają konstrukcje typu `sync_synchronize()`. W Assembly x86_64 mamy prefix `lock`, operacje atomowe, memory barriers (`lfence` – load, odczyt nie może wykonać się przed tą instrukcją, `sfence` – store, zapis nie może być po, `mfence` – oba) oraz operacje serializujące (np. `cpuid`), które powodują synchronizację całego procesora (i są bardzo drogie).

Istnieją instrukcje read-modify-write, które wykonują te trzy rzeczy naraz. Zwykłe operacje bitowe i dodawanie/odejmowanie tak działają. Ważna jest operacja porównująca `cmpxchg` – porównujemy `rax` z pierwszym argumentem, jak są równe wsadzamy drugi argument do pierwszego, jak różne wsadzamy pierwszy do `rax`. Mamy też instrukcję `xchg`, która zamienia zawartość dwóch argumentów atomowo.

W C++ mamy typ `atomic`, który trzyma wartość i pozwala robić na niej atomowe instrukcje. Mamy `store`, `load`, `exchange`, a do tego `compare_exchange_strong`, który działa jak `cmpxchg` i zwraca, czy zamiana się udała. `compare_exchange_weak` robi to samo, ale może zwrócić nieudaną zamianę nawet jak się uda. Te funkcje przyjmują parametr typu `memory_order`, który określa kolejność wykonania względem innych instrukcji i ma wartości:

- `memory_order_relaxed` – nie ma wpływu na kolejność.
- `memory_order_acquire` i `memory_order_release` – jeśli read z acquire zobaczył zapis z release, to wszystkie późniejsze odczyty na pewno zobaczą wszystkie zapisy, które były przed tym zapisem (niezależnie od danych, których dotyczą). Da się to zaimplementować za pomocą odpowiedniej instrukcji `fence`. Mamy gwarancję tylko co do późniejszych odczytów i tylko, jeśli implikacja jest spełniona.
- `memory_order_consume` – to samo, ale tylko dla danych zależnych.
- `memory_order_acq_rel` – oba jednocześnie, jedno tyczy się części instrukcji, która czyta, a drugie tej, która pisze.
- `memory_order_seq_cst` – wszystkie wątki używające tego porządku są sequentially consistent, czyli istnieje spójna kolejność wydarzeń na wszystkich tych wątkach.

To wszystko pozwala nam robić nieblokujące struktury danych. Będą to struktury, na których pracuje wiele wątków, a spójność struktury jest zapewniana bez blokowania wątków. Zatrzymywanie wątków jest drogie, bo wymaga odwołania do systemu. Do tego może nastąpić priority inversion – jeśli wątek z małym priorytetem przejmie sekcję krytyczną a wątek z dużym priorytetem się na niej zablokuje, to wątek ze średnim priorytetem może przejąć procesor od tego z małym i wtedy wątek z dużym priorytetem czeka na ten ze średnim.

Nieblokujące struktury danych są projektowane tak, aby operacje były serializowalne. To znaczy, że struktura zachowuje się tak, jakby każda operacja wykonała się atomowo w jakimś punkcie czasu pomiędzy jej rozpoczęciem a zakończeniem. Sama operacja nie jest atomowa, ale będzie istniała kolejność operacji, w której operacje wyglądają jak atomowe. Do tego mamy własność globalnego postępu – w każdej sytuacji przynajmniej jeden działający wątek odnotuje postęp w skończonej liczbie kroków.

Możemy zaimplementować lock-free stos w następujący sposób.

```
template<typename Data>
class LockFreeStack {
    struct Node {
        Data data; Node* next;
    };
    Node* top;

    void push(Data data) {
        Node* n = new Node();
        n->data = data;
        do {
            n->next = top;
        } while (!CAS(&top, n->next, n));
   }

   Data pop() {
        Node* n;
        do {
            n = top;
            if (!n) throw StackUnderflow()
        } while (!CAS(&top, n, n->next));
        Data data = n->data;
        delete n;
        return data;
   }
}
```

CAS oznacza instrukcję compare and swap działającą jak `compare_exchange`. W tej implementacji są problemy. `n = top;` nie jest atomowe i może się przepleść z atomowym CAS. Rozwiązaniem jest użycie `atomic`. Mechanizm zarządzania pamięcią nie jest lock-free – można statycznie prealokować pamięć lub mieć własny alokator (np. drugi stos). Podobnie jest z wyjątkami – można z nich zrezygnować lub zaakceptować ten przypadek brzegowy.

Występuje problem ABA – jeden wątek przesuwa do zmiennej szczyt stosu, drugi usuwa go i dodaje dwa nowe elementy, ostatni taki sam jak początkowy szczyt (o takim samym adresie). Następnie pierwszy wątek przesuwa szczyt stosu do zapisanego przedtem poprzednika. W ten sposób stracimy dwa elementy, a instrukcja CAS nam nie pomoże, bo wartość będzie taka sama. Aby się przed tym bronić można wersjonować zmienne (zmieniamy zapisaną wersję przy zmianie wartości). Może też zdarzyć się tak, że jeden wątek zapisze pamięć do zmiennej, a drugi zwolni obiekt pod tą pamięcią. Wtedy pierwszy wątek odwoła się do nieistniejącej pamięci. Dlatego istnieje hazard pointer – każdy wątek trzyma listę używanych (niebezpiecznych) wskaźników do pamięci. Wątki nie zwalniają pamięci, jeśli znajduje się na liście innego wątka. Odkładają wskaźniki na specjalną listę, z której elementy są dealokowane, gdy nikt ich nie używa.

Z punktu widzenia wydajności przy niskim congestion (mało wątków walczy o sekcje krytyczne) zmiany kontekstu (zatrzymywanie wątków z pomocą systemu) są drogie, a operacje atomowe tanie (bo zazwyczaj można po prostu wykonać operację, nikt inny jej nie wykonuje). Przy wysokim congestion operacje atomowe stają się drogie – wtedy opłaca się nam blokować wątki. Do zaimplementowania tego może nam posłużyć futex (fast userspace mutex), który z punktu widzenia użytkownika jest zwykłą liczbą, a w kernelu jest to kolejka wątków czekających na zadanym adresie. Formalnie każdy adres ma swój futex, w praktyce większość nie istnieje. W Linuxie jest syscall `futex`, która przyjmuje adres, o który chodzi, numer operacji i wartość. `FUTEX_WAIT` usypia wątek, jeśli wartość jest taka jak adres. `FUTEX_WAKE` budzi maksymalnie tyle wątków, ile wynosi wartość. Za pomocą futexów można zaimplementować mutexy – mamy atomową zmienną, która oznacza przejęcia danych. Jeśli wiele wątków walczy o dane, to zasypiamy, jeśli nie, to robimy lock-free pętle.

11. Obsługa I/O

2025-05-28

Urządzenia I/O można podzielić ze względu na szybkość: wolne to takie, które dostarczają dane wolniej niż CPU może je przetwarzać. Średnio szybkie to takie lekko wolniejsze od CPU, szybkie są szybsze od CPU. Urządzenia mogą modyfikować swoją prędkość poprzez buforowanie (sprzętowe) danych – można zbuforować dużo i wysłać wszystko naraz (szybko) lub buforować nadmiar i przesyłać wolno.

Urządzenia dzielimy też ze względu na obsługę gotowości do przekazania danych: polled (wymagają sprawdzenia gotowości), asynchronous (informują o gotowości poprzez przerwania), sampling (zawsze gotowe).

Podstawową komunikację mamy przez port-mapped I/O (PIO). Mamy przestrzeń portów (64kB), która jest odrębna od przestrzeni adresów, porty mogą być zmapowane do fizycznych elementów urządzenia. Mamy porty read-only, write-only, read-write i dual (w dual pisanie i czytanie odbywa się do różnych portów). Mamy dedykowane instrukcje `in` i `out`, odczyt i zapis wymaga wielu operacji procesora (pętla do przesyłania pojedynczych słów). Dlatego działa to tylko dla wolnych urządzeń. Do tego portów jest dość mało. Polling jest możliwy – mamy porty, które dają informację o gotowości.

Szybsze jest memory-mapped I/O (MMIO). Mamy już dużą przestrzeń adresów i szynę danych do komunikacji z RAM'em, można do niej podłączyć dowolne urządzenie. Urządzenie jest zmapowane do fragmentu przestrzeni adresowej. Nie potrzebujemy odrębnej magistrali adresów i danych, łatwo współdzielić urządzenie (jak w RAM'ie), mamy te same optymalizacje I/O, co w przypadku RAM'u. Do tego przepisywanie danych między częściami przestrzeni adresowej jest prostsze niż do portów. Mimo to nie możemy obsługiwać szybkich urządzeń. Polling jest możliwy podobnie jak przy PIO. Wymagana jest kontrola zachowania cache'u – jeśli urządzenie daje dostęp np. do obrazu w czasie rzeczywistym, to nie chcemy go cache'ować. Cache może być wyłączony, działać w trybie write-back (do cache'u, a po jakimś czasie do RAM'u), write-through (do RAM'u, być może do cache'u), write-combining (grupowanie zapisów do tej samej linii cache'u). W danym momencie często mamy zmapowany tylko fragment urządzenia (np. dysku), aby dostać się do innych danych musimy poinformować o tym urządzenie (np. poprzez PIO) i poczekać na jego gotowość.

Direct Memory Access (DMA) polega na bezpośredniej komunikacji urządzenia z pamięcią. Normalnie mamy magistralę, do której jest podpięty procesor, RAM i urządzenia, a procesor wysyła polecenia zapisu i odczytu. Możemy pozwolić innym urządzeniom na wydawanie takich poleceń, które RAM będzie wykonywać – pisać w odpowiednie miejsce pamięci. Nie wymaga to udziału CPU, a więc umożliwia obsługę szybkich urządzeń. Nie jest możliwy polling (bo procesor nie komunikuje się z urządzeniem). Do tego potrzebne są protokoły zarządzania szynami adresów i danych (bus arbitration protocols). Jest to

na tyle dobra metoda, że chcemy móc ją stosować nawet z urządzeniami, które jej nie wspierają. Do tego służą silniki DMA. Są to programowalne urządzenia do transferu danych, które w komunikacji z innymi urządzeniami przejmują rolę CPU i umożliwiają transfer bez jego udziału. Umożliwiają też transfery bezpośrednio między urządzeniami.

Zazwyczaj stosuje się rozwiązania hybrydowe. Kontrola stanu urządzania zazwyczaj odbywa się poprzez PIO lub MMIO, sam transfer danych za pomocą DMA. Silnik DMA jest programowany poprzez PIO lub MMIO. Komunikacja odbywa się za pomocą pollingu lub poprzez asynchroniczne przerwania.

W Linux'ie wszystkie urządzenia są widoczne jako pliki („devices") i mają interfejs plików – wykorzystują file descriptory, są syscalle `open, close, read, write`, które robią PIO lub pośrednie MMIO (kernel wykonuje MMIO za nas). `mmap` to bezpośrednie MMIO – mapujemy urządzenie na naszą przestrzeń adresową. Specjalne instrukcje dla urządzeń wykonujemy syscallem `ioctl` (zazwyczaj PIO). Mamy dostępny polling, przerwania opakowane w sygnały oraz (niemożliwe na niższym poziomie) odczyty i zapisy synchroniczne (system zatrzymuje proces, aż odczyt będzie możliwy).

Urządzenia zazwyczaj dzielimy na znakowe (character devices) oraz blokowe (block devices). Urządzenia znakowe umożliwiają odczyt i zapis po jednym bajcie, zazwyczaj nie buforują danych i są urządzeniami strumieniowymi – pozwalają na dostęp sekwencyjny (nie działa `seek`, czyli przesuwanie descriptora) i nie można na nich wykonać `mmap`. Urządzenia blokowe zapewniają odczyt w blokach (dyski mają zazwyczaj 512B, nie jest możliwy dostęp do np. jednego bajtu), następuje buforowanie danych (zwłaszcza przy pisaniu). Mamy swobodny dostęp do danych, często możliwe jest wykonanie `mmap`.

Obsługa zdarzeń sprzętowych może być synchroniczna poprzez polling – mamy wtedy prosty system bez synchronizacji, ale marnujemy czas procesora i możemy dostać informację tylko o jednym zdarzeniu naraz. Obsługa asynchroniczna (poprzez sygnały) jest bardziej elastyczna i mniej marnuje zasoby, ale jest bardziej skomplikowana.

Przerwania sprzętowe mogą być generowane przez procesor (automatycznie lub instrukcją `int`) lub przez urządzenia. Przerwania mają jednobajtowe identyfikatory, które informują o przyczynie przerwania. Czasem nie chcemy, aby przerwanie się wydarzyło (czasem np. system zmienia funkcje obsługujące przerwania i nie chce, aby wtedy wykonało się przerwanie). Do globalnego włączania i wyłączania przerwań służą instrukcje `cli` i `sti` (ustawiają odpowiednią flagę). Do poszczególnych przerwań trzeba skorzystać z Advanced Programmable Interrupt Controller. Same przerwania są obsługiwane asynchronicznie za pomocą odpowiednio zdefiniowanych funkcji (Interrupt Service Routine). Następuje context switch, podczas którego stan procesora jest odkładany na stos. Zazwyczaj przerwanie jest obsługiwane z uprawnieniami systemu.

W Linux'ie syscalle `read` i `write` domyślnie blokują się w oczekiwaniu na gotowość urządzenia, na deskryptorach otwartych z `O_ASYNC` zwracają błąd zamiast się blokować. Syscall `select` umożliwia czekanie na gotowość któregoś z deskryptorów. Jest to stary syscall, lepiej korzystać z `epoll_wait`, który daje możliwość grupowania wielu deskryptorów w jeden, dynamicznego zarządzania ich zbiorem i oczekiwania na gotowość w trybie edge-triggered (na przyjście danych) i level-triggered (na obecność danych).

Sygnały to programowy odpowiednik przerwań, przerwania są wytwarzane przez procesor, a sygnały przez programy z pomocą systemu. Sygnały są skierowane do konkretnych procesów (lub wątków) i są obsługiwane asynchronicznie przez zarejestrowane funkcje. Sygnały mogą opakowywać przerwania błędów procesora (`SIGSEGV`, `SIGFPE`, `SIGILL`, `SIGTRAP` – ten ostatni pojawia się przy debugowaniu i jest wysyłany co instrukcję Assemblera), informować o zdarzeniach (`SIGCHLD`, `SIGINT`, `SIGHUP`) lub być wysłane ręcznie przez syscall `kill` lub `sigqueue`.

Do obsługi sygnałów wewnątrz procesu może służyć syscall `signal`, który jest deprecated i ma niespójną semantykę w różnych systemach – ma niezdefiniowane zachowanie na programach wielowątkowych i można przerwać funkcję obsługującą sygnał tym samym sygnałem. Dlatego należy go używać tylko do ignorowania sygnałów i przywracania domyślnej obsługi. Pozostałe rzeczy robimy `sigaction` – rejestrujemy funkcje obsługi, które dostają dodatkowe informacje o źródle sygnału, a do tego wybrane sygnały są maskowane podczas obsługi sygnału (w szczególności on sam).

Syscall `sigtimedwait` umożliwia synchroniczne czekanie na sygnał z danego zbioru. Syscall `signalfd` opakowuje zbiór sygnałów w file descriptor, możemy odebrać jeden z nich synchronicznie za pomocą `read` lub czekać łącznie z prawdziwymi deskryptorami za pomocą `poll`, `select`.

Sygnały mogą być buforowane przez system, zatem wiele sygnałów może się nałożyć – nie możemy na

nich polegać do przekazania jakiejś informacji. Istnieją sygnały niezawodne (reliable, real-time). Mają wyznaczony zakres numerów (od `SIGRTMIN` do `SIGRTMAX`), wysłane sygnały są kolejkowane (funkcja obsługi wykona się dokładnie raz dla każdego wysłania), wysłanie może się nie udać, jak kolejka jest pełna (wtedy wysyłający widzi błąd). Występuje priorytetyzowanie sygnałów – te o niższym numerze będą obsługiwane wcześniej. Do tego wysłanie sygnału real-time za pomocą `sigqueue` pozwala na wysłanie niewielkiej ilości danych (8 bajtów w x86_64).

Sygnały można blokować. Nie trafiają wtedy do procesu, fakt wysłania sygnału czeka, aż zostanie odblokowany. Maskę zablokowanych sygnałów zmieniamy za pomocą `sigprocmask`. Można też czekać na ustalonej masce za pomocą `pselect`. Podczas obsługi sygnału stan procesora jest odkładany na stos (lub w miejsce zdefiniowane przez `sigaltstack`). Sygnały mogą nastąpić podczas wykonywania funkcji z biblioteki standardowej, dlatego podczas obsługi sygnału możemy wykonywać niewiele spośród jej funkcji. Aby obsługa sygnału mogła komunikować się z resztą programu potrzebne są wspólne dane zadeklarowane jako `volatile` (aby program na pewno zobaczył zmianę bo sygnale). Jeśli sygnał przyjdzie w trakcie syscalla, to niektóre syscalle (szybkie) wykonają się i dopiero wtedy nastąpi przerwanie, a pozostałe (wolne) przerwą swoje wykonanie, zwracając `EINTR` lub częściowe wyniki. W `sigaction` jest flaga `SA_RESTART`, która powoduje restart syscalla przerwanego przez odpowiedni sygnał.

Pojawienie się pilnych danych w urządzeniu I/O może wywołać sygnał `SIGURG`. Podczas pracy z terminalem pojawiają się sygnały `SIGTTIN`, `SIGTTOU`. Operacje na deskryptorach z ustawioną przez `fcntl` flagą `F_SETSIG` generują wybrany sygnał (dla urządzeń, nie dla zwykłych plików).

Podczas obsługi sygnałów w programie wielowątkowym trzeba pamiętać o tym, że funkcje obsługi są wspólne dla wszystkich wątków, ale maski blokowania oddzielne. `sigwait` wykonany w wątku wyłącza obsługę swoich sygnałów w pozostałych wątkach – nie ma sensu, żeby inny wątek zareagował na jeden z tych sygnałów, jeśli ten wątek na nie czeka. `kill` i `sigqueue` wysyłają sygnał do całego procesu, obsługuje go którykolwiek wątek. Sygnały zabijające powodują śmierć całego procesu. `pthread_kill` wysyła sygnał do konkretnego wątku.