Architektura Komputerów i Systemy Operacyjne

nieoficjalne kompendium, wersja 1.0

K. Kleczkowski, M. Pietrek, J. Nigiel, Sz. Wróbel,
prof. dr hab. inż. A. Lasecki, D. Nowak, J. Gogola

29 stycznia 2018

Spis treści

Spis tablic

Wprowadzenie

To jest naprawdę bardzo proste!

(dr Marcin Zawada)

Poniższy skrypt powstał jako odpowiedź na nadchodzący egzamin z Architektury Komputerów i Systemów Operacyjnych. Treść jest ukierunkowana do studentów informatyki, którzy znają już takie podstawy jak systemy pozycyjne, programowanie w języku C, czy pobieżne korzystanie z Linuksa.

Rozdział 1 autorstwa Jarosława Nigiel opisuje systemy liczbowe, w tym systemy pozycyjne, zamianę systemów pozycyjnych, operacje arytmetyczne uogólnione na dowolne systemy pozycyjne.

Rozdział 2 autorstwa Jarosława Nigiel opisuje reprezentację liczb, w tym reprezentację w kodzie uzupełnieniowym oraz reprezentację liczb zmiennoprzecinkowych wg standardu IEEE-457.

Rozdział 3 autorstwa Jarosława Nigiel oraz Mikołaja Pietrka opisuje algebrę Boole'a, jej własności, analogię między rachunkiem zdań i teorią mnogości oraz wprowadza Czytelnika do funkcji boolowskich i metod ich minimalizacji, to znaczy, metodę Karnaugha i metodę Quine'a-McCluskey'a.

Rozdział 4 autorstwa Mikołaja Pietrka opisuje układy kombinacyjne, problem hazardu, przykładowe układy oraz ich zastosowanie.

Rozdział 5 autorstwa Mikołaja Pietrka mówi o układach sekwencyjnych, przykładowych układach oraz ich zastosowaniach.

Rozdział 6 autorstwa Mikołaja Pietrka opowiada o odpowiednich architekturach procesorów ich charakterystycznych własnościach i przykładowych przedstawicielach danych architektur.

Rozdział 7 autorstwa ??? mówi o rodzajach pamięci, sposobie ich realizacji, miejscu występowania i zastosowaniu.

Rozdział 8 autorstwa ??? opisuje techniki, jakie procesory wykorzystują do zarządzania pamięcią i jej ochroną. Również opowiada o podziale pamięci i pamięci skojarzeniowej procesora (ang. *cache*).

Rozdziały 9, 10 i 11 autorstwa Aleksandra Laseckiego, Damiana Nowaka i Szymona Wróbla mówią o programowaniu w assemblerze na konkretne platformy, notacji asemblera, trybach adresowania, sposobie realizacji stosu, konwencji wywołań, wybranych rozkazów procesora. Dodatkowo w rozdziale o x86 wprowadzone są pojęcia trybu rzeczywistego i chronionego, jego realizacji. Również jest wspomniane o koprocesorze x87 i jego zasadzie działania.

Rozdział 12 autorstwa Szymona Wróbla opowiada o API do wykorzystania karty graficznej do obliczeń rozproszonych — CUDA. Mówi o architekturze wielowątkowej oraz zwięźle opisuje działanie przykładowego programu napisanego w 'CUDA C'.

Rozdział 13 autorstwa Konrada Kleczkowskiego opisuje podstawy realizacji wielozadaniowości w systemach uniksopodobnych, wprowadza pojęcie procesu i wątku. Również opisy zawierają informacje na temat procesów zombie, IPC, czy zarządzaniu procesami potomnymi. Rozdział również opisuje programowanie wielowątkowe, wprowadzając mechanizmy synchronizacji wątków.

Rozdział 14 autorstwa Konrada Kleczkowskiego opisuje podstawy obsługi wejścia-wyjścia i zarządzania plikami z perspektywy powłoki użytkownika oraz wywołań systemowych. Wprowadza również do realizacji komunikacji sieciowej w systemach uniksopodobnych.

Rozdział 15 autorstwa Aleksandra Laseckiego opisuje zarządzanie pamięcią w systemach uniksopodobnych. Wprowadza pojęcie stosu i sterty. Opisuje w jaki sposób system operacyjny zarządza stertą programu.

Rozdział 16 autorstwa Jakuba Gogoli opisuje pliki wykonywalne i biblioteki dynamiczne w formacie ELF. Opisuje podstawową budowę pliku ELF i mechanizm relokowania kodu.

Rozdział 17 autorstwa Jakuba Gogoli opisuje podstawową budowę systemu plików FAT i podstawowe zachowanie tego systemu plików.

Rozdział 18 autorstwa Konrada Kleczkowskiego mówi o języku C, podstawowych typach i algorytmie wyrównywania struktur.

Podziękowania

Dziękujemy dr. Marcinowi Zawadzie oraz wszystkim prowadzącym laboratoria i ćwiczenia z Architektury Komputerów i Systemów Operacyjnych w semestrze zimowym 2017/2018. Szczególne podziękowania należą się mgr. Ilyi Hradovichowi oraz dr. Przemysławowi Błaśkiewiczowi za przekazanie wiedzy wykraczającej poza program kursu oraz czas poświęcony na laboratoriach.

Część I

Architektura komputerów

Rozdział 1

Systemy liczbowe

Liczby rządzą światem.

(Pitagoras)

1.1 Systemy pozycyjne

Każda liczba jest reprezentowana w następujący sposób:

$$c_{n-1}p^{n-1} + ... + c_2p^2 + c_1p^1 + c_0p^0 = \sum_{i=0}^{n-1} c_ip^i$$

gdzie p jest nazywane **podstawą**, $0 \leqslant c_i \leqslant p-1$ nazywane jest cyfrą, a n to jest ilość cyfr. Liczby takie można przedstawić również w wersji skróconej:

$$(c_{n-1}...c_2c_1c_0)_p$$

lub jeśli podstawa jest wiadoma:

$$c_{n-1}...c_2c_1c_0$$

Taki sposób reprezentacji liczby nazywamy **systemem pozycyjnym**.
Najczęściej używane przez nas podstawy to 2, 10 i 16.
Przykłady:

$$(75)_{10} = 7 * 10 + 5 * 1$$
$$(111000)_2 = 1 * 2^5 + 1 * 2^4 + 1 * 2^3$$
$$(AB)_{16} = A * 16 + B * 1$$

1.2 Zamiana podstaw systemów pozycyjnych

1.2.1 Liczby całkowite

Sposób I (dzielenie):
Aby zmienić podstawę liczby, należy starą liczbę dzielić przez podstawę, zapisując resztę do najmniej znaczącej cyfry, tak długo póki nie otrzymamy 0.

Przykłady:

$$(23)_{10} = (?)_2$$

$$23 : 2 = 11 \; r \; \mathbf{1}$$

$$11 : 2 = 5 \;\; r \; \mathbf{1}$$

$$5 : 2 = 2 \;\; r \;\; \mathbf{1}$$

$$2 : 2 = 1 \;\; r \; \mathbf{0}$$

$$1 : 2 = 0 \;\; r \; \mathbf{1}$$

Ponieważ zaczynamy reszty od najmniej znaczącej cyfry, mamy:

$$(23)_{10} = (10111)_2$$

$$(231)_{10} = (?)_{16}$$

$$231 : 16 = 14 \; r \; \mathbf{7}$$

$$14 : 16 = 0 \; r \; \mathbf{14 \; (E)}$$

$$(231)_{10} = (E7)_{16}$$

Sposób II (algorytm „zachłanny"):
Bierzemy największą możliwą potęgę nowej podstawy, zapisujemy ją w systemie o starej podstawie, następnie odejmujemy od starej liczby, zapisując na najbardziej znaczącej cyfrze ile razy odejmowaliśmy. Powtarzamy algorytm aż do otrzymania 0.
Przykład:

$$(123)_{10} = (?)_2$$

$$123 \geqslant 2^6 = 64$$

$$123 - \mathbf{1} * 64 = 59 \geqslant 2^5$$

$$59 - \mathbf{1} * 32 = 27 \geqslant 2^4$$

$$27 - \mathbf{1} * 16 = 11 \geqslant 2^3$$

$$11 - \mathbf{1} * 8 = 3 \leqslant 2^2$$

$$3 \geqslant 2^1$$

$$3 - \mathbf{1} * 2 = 1 \geqslant 2^0$$

$$1 - \mathbf{1} * 1 = 0$$

Nie odejmowaliśmy jedynie liczby 2^2, na tej pozycji mamy 0.

$$(123)_{10} = (1111011)_2$$

Teraz przykład nieco trudniejszy:

$$(215)_7 = (?)_4$$

$$4^4 = (514)_7$$

$$215 \geqslant 4^3 = (121)_7$$

$$215 - \mathbf{1} * 121 = 64 \geqslant 4^2 = 22$$

$$64 - \mathbf{2} * 22 = 20 \geqslant 4^1$$

$$20 - \mathbf{3} * 4 = 2 \geqslant 1$$

$$2 - \mathbf{2} * 1 = 0$$

$$(215)_7 = (1232)_4$$

1.2.2 Zamiana systemów, których podstawy są swoimi potęgami

Jeśli nowa podstawa (p_{new}) jest w postaci $p_{new} = p_{old}^a$, to w liczbie, którą chcemy zamienić, grupujemy od prawej po a cyfr, i każda z tych grup tworzy jedną cyfrę w nowym systemie. Jeśli najstarsza grupa jest niepełna, dopisuje się zera.
Przykłady:

$$(100011)_2 = (?)_{16}$$

$$(0010|0011)_2 = (2|3)_{16} = (23)_{16}$$

$$(1022201)_3 = (?)_9$$

$$(01|02|22|01)_3 = (1|2|8|1)_9 = (1281)_9$$

Jeśli stara podstawa jest w postaci $p_{old} = p_{new}^a$, to zamieniamy cyfry na nowy system, dopisując zera po lewej stronie tak, aby stworzyć grupę a-elementową. **Przykłady:**

$$(D1)_{16} = (1110|1)_2 = (1110|0001)_2 = (11100001)_2$$

$$(3781)_9 = (10|21|22|01)_3 = (10212201)_3$$

1.2.3 Liczby wymierne

Aby zamienić podstawę liczby wymiernej, należy część całkowitą zamienić normalnie, a następnie część ułamkową mnożyć przez nową podstawę, zapisując część całkowitą z wyniku na kolejnej pozycji po przecinku.
Przykłady:

$$(2.25)_{10} = (?)_2$$

$$(2)_{10} = (10)_2$$

$$0.25 * 2 = \mathbf{0}.5$$

$$0.5 * 2 = \mathbf{1}.0$$

$$(2.25)_{10} = (10.01)_2$$

$$(3.4)_{10} = (?)_3$$

$$(3)_{10} = (10)_3$$

$$0.4 * 3 = \mathbf{1}.2$$

$$0.2 * 3 = \mathbf{0}.6$$

$$0.6 * 3 = \mathbf{1}.8$$

$$0.8 * 3 = \mathbf{2}.4$$

0.4 się powtarza, więc otrzymaliśmy ułamek okresowy.

$$(3.4)_{10} = (10.(1012))_3$$

1.3 Operacje na liczbach w systemie pozycyjnym

Operacje na liczbach w dowolnym systemie pozycyjnym wykonuje się w taki sam sposób jak w systemie dziesiętnym, najlepiej pisemnie.

Przykład:

$$(234)_{16} * (5)_{16} = (4 * 5) * 16^0 + (3 * 5) * 16^1 + (2 * 5) * 16^2$$

$$(234)_{16} * (5)_{16} = 4 * 16^0 + (1 + E) * 16^1 + A * 16^2$$

$$(234)_{16} * (5)_{16} = 4 * 16^0 + 0 * 16^1 + B * 16^2 = (B04)_{16}$$

Rozdział 2

Reprezentacja liczb

2.1 Naturalny kod binarny

W naturalnym kodzie binarnym wszystkie liczby mają ustaloną długość, a jeśli przy konwersji liczby z dziesiętnego na binarny liczba ma mniej cyfr, uzupełnia się ją zerami z przodu. Przykład:

Tablica 2.1: NKB 3-bitowe

n	NKB(n)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

2.2 Przesunięty kod binarny (ang. *biased*)

Schemat generowania n-bit biased representation jest następujący:

$$(0)_{10} = 1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} 0 \cdot 2^i$$

Pozostałe liczby są generowane poprzez dodanie lub odjęcie 1 w obu systemach (binarnym i dziesiętnym) Przykład 3-bitowy:

Tablica 2.2: Biased 3-bitowe

n	biased binary
-4	000
-3	001
-2	010
-1	011
0	100
1	101
2	110
3	111

2.3 Kod uzupełnieniowy do jedności (U1)

Jeżeli liczba, którą chcemy zapisać w n-bitowym U1 jest dodatnia, to najstarszy bit jest równy 0, a pozostałe bity tworzą (n-1)-bitowe NKB. Jeżeli liczba jest ujemna, to najstarszy bit jest równy 1, następnie moduł z tej liczby jest zapisywany w (n-1) bitowym NKB. Wynik tej operacji jest zamieniany w taki sposób, że jedynki tworzą zera a zera jedynki.

Przykład:

$$U1_8((-73)_{10}) =?$$

$$-73 < 0$$

$$(73)_{10} = (1001001)_2 -> 0110110$$

$$U1_8(-73) = 10110110$$

UWAGA. Zauważmy, że w U1 jest zero ujemne i zero dodatnie. (1...1 i 0...0)

2.4 Kod uzupełnienowy do dwójki (U2)

Liczby dodatnie w U2 przedstawione są w taki sam sposób jak NKB Liczby ujemne w U2 powstają w następujący sposób:

- Niech a<0, n - liczba bitów w U2
- Ustawiamy najstarszy bit na 1
- Pozostałe bity to n-1 bitowe NKB z liczby $2^{n-1} + a$

Przykład:

$$U2_8((-73)_{10}) =?$$

$$-73 < 0$$

$$2^{8-1} + (-73) = 128 - 73 = 55 = (0110111)_2$$

$$U2_8((-73)_{10}) = 10110111$$

Spójrzmy na tę tabelę:

Tablica 2.3: Różnica między U1 a U2

reprezentacja binarna	U1	U2
000	+0	0
001	1	1
010	2	2
011	3	3
100	-3	-4
101	-2	-3
110	-1	-2
111	-0	-1

Można zauważyć, że w liczbach dodatnich $U1 = U2$, a w ujemnych $U1 = U2 + 1$

2.5 Liczby zmiennoprzecinkowe

2.5.1 Liczby w standardzie IEEE-754

Liczby w standardzie IEEE-754 są przedstawione w następujący sposób:

- najstarszy bit jest bitem znaku (S)
- kolejne 8 (11 dla double) bitów to wykładnik (E)
- ostatnie 23 (52 dla double) bity nazywane są mantysą (M)

Jeśli chcemy dokładnie wiedzieć, jaką wartość przedstawia liczba w formacie IEEe-754, należy wykonać następujące obliczenia:

$$E! = 0...0 \wedge M \neq 1...1 \Rightarrow (-1)^S(1, M)_2 2^{E-B} \text{(liczba znormalizowana)}$$

$$E = 0...0 \wedge M \neq 0...0 \Rightarrow (-1)^S(1, M)_2 2^{1-B} \text{ (liczba zdenormalizowana)}$$

$$E = 0...0 \wedge M = 0...0 \Rightarrow 0$$

$$E = 1...1 \wedge M = 0...0 \Rightarrow (-1)^S \infty$$

$$E = 1...1 \wedge M \neq 0...0 \Rightarrow NaN \text{ (Not a Number)}$$

B jest stała, dla float równa 127, dla double 1023.
Przykład:

$$n = 11000000111100110101101010110101$$

n jest 32-bitowe, więc mamy do czynienia z float.

$$S = 1$$

$$E = 10000001 = (129)_{10}$$

$$M = 11100110101101010110101$$

$$(1.M)_2 = \left(1 + \frac{7559861}{8388608}\right)_{10}$$

$$n = -\left(1 + \frac{7559861}{8388608}\right) \cdot 2^{129-127} = -7.604822635650634765625$$

2.5.2 Zamiana wymiernego naturalnego kodu binarnego na liczbę w formacie IEEE-754

Jeśli liczba binarna jest w postaci $\pm b_n b_{n-1}...b_0, c_0...c_k$, to należy przesunąć przecinek w lewo (tak jak się to robi w przypadku notacji wykładniczej). Wtedy liczba będzie wyglądała w taki sposób:

$$\pm b_n, b_{n-1}...b_0 c_0...c_k \cdot 2^{n-1}$$

Aby teraz zamienić taką liczbę na IEEE-754, wystarczy zauważyć, że:

$$M = b_{n-1}...b_0 c_0...c_k 0...0$$

Liczba zer na końcu jest równa (23 lub 52) - $(n-1) - k$

$$E = NKB((n-1) + (127 lub 1023))$$

Przykład:
Jak można zapisać $(-7.375)_{10} = -111.011$ w IEEE-754 (float)?

$$111.011 = 1.11011 \cdot 2^2$$

$$S = 1$$

$$M = 11011000000000000000000$$

$$E = 2 + 127 = 129 = (10000001)_2$$

Nasza liczba zatem to:

$$11000000111011000000000000000000 = (C0EC0000)_{16}$$

2.5.3 Prównywanie liczb w formacie IEEE-754

Algorytm porównywania liczb w IEEE-754 jest następujący (autorski):

- weźmy a, b w formacie IEEE-754
- jeśli $S(a) \neq S(b)$, to mniejsza jest liczba której $S = 1$. Następne kroki dla ułatwienia zapisu przedstawię tylko dla liczb dodatnich, dla liczb ujemnych jest dokładnie odwrotnie
- jeśli $E(a) = E(b)$, to większa jest liczba o większej mantysie
- jeśli wykładniki są rózne to mamy p...., a nie, to też jest proste. Wystarczy wtedy porównać wykładniki. Dowód pozostawiam czytelnikowi, bo jest bardzo prosty.

Rozdział 3

Algebra Boole'a

Tyle jest w każdym poznaniu nauki, ile jest w nim matematyki.

(Immanuel Kant)

3.1 Definicja algebry Boole'a

Algebra Boole'a jest algebrą $(S, +(x,y), *(x,y), '(x), 0_S, 1_S)$, w którym S jest zbiorem, 0_S i 1_S są stałymi na tym zbiorze, a działania na stałych zdefiniowane są następującą tabelką:

Tablica 3.1: Działania w algebrze bóla

x	y	+(x,y)	*(x,y)	'(x)
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

W dalszej części skryptu będziemy używać nieformalnej i ogólnie przyjętej notacji $+(x,y) \equiv x + y$, $*(x,y) \equiv xy$, $'(x) \equiv x'$.
Ponadto, każda algebra Boole'a musi spełniać następujące aksjomaty:

- $(\forall x)\,(x + 0 = x)$
- $(\forall x)\,(x * 1 = x)$
- $(\forall x, y)\,(x + y = y + x \wedge xy = yx)$
- $(\forall x, y, z)\,(x + (y + z) = (x + y) + z)$
- $(\forall x, y, z)\,(x\,(yz) = (xy)\,z)$
- $(\forall x, y, z)\,(x + (yz) = (x + y)\,(x + z))$

- $(\forall x, y, z)\,(x\,(y + z) = (xy) + (xz))$
- $(\forall x)\,(xx' = 0 \wedge x + x' = 1)$

3.1.1 Przykładowe Algebry Boole'a

Przykładowymi algebrami Boole'a są:

- $(\{false = 0, true = 1\}, AND, OR, NOT, 0, 1)$
- $(\mathbb{P}(A), \cup, \cap, \emptyset, A)$ (gdzie A jest dowolnym zbiorem, a $\mathbb{P}(A)$ to powerset)

Obydwa przykłady są bardzo często używane w informatyce. Powinno się pokazać, że to są rzeczywiście algebry Boole'a (spełniają aksjomaty), dowód pozostawiam czytelnikowi.

3.2 Funkcja boolowska

Funkcją boolowską jest każda funkcja $f : \{0,1\}^n \to \{0,1\}$. Każde zdanie logiczne może definiować inną funkcję boolowską. Na przykład $f(x, y) = x \leftrightarrow y$ jest funkcją boolowską.
Funkcje boolowskie można opisać na kilka różnych sposobów. Jednym z nich jest tabelka prawdy. Na przykład:

Tablica 3.2: Tabelka prawdy

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Tą samą funkcję można również zapisać tak jak wcześniej, w postaci zdania logicznego. Robimy to na dwa sposoby. Pierwszy z nich polega na tym, że tam gdzie w wyniku funkcji występuje 1, (np dla f(0,0,0)), „koduje się" zmienne które powodują tę wartość za pomocą koniunkcji tak, aby otrzymać 1. Następnie wszystkie te „kody" łączy się alternatywą. Wynik jest następujący:

$$f(x, y, z) = (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge \neg z)$$

Ponieważ operujemy w algebrze Boole'a, dla uproszczenia zapisu można stosować zamiast znaków znanych nam z logiki, znaki +,* oraz '. Wtedy ta sama funkcja będzie wyglądała następująco:

$$f(x, y, z) = x'y'z' + x'yz + xy'z'$$

Ten zapis znany jest również pod nazwą SOP (ang. Sum Of Products). Innym znanym zapisem funkcji boolowskich jest tzw. POS (ang. Product of Sums). Generuje się go w taki sposób, że „koduje się" tym razem

sumy tych zmiennych, które dają 0. Powstałe kody łączy się za pomocą produktu. Dla naszej funkcji będzie miało to postać:

$$f(x, y, z) = (x + y + z')(x + y' + z)(x' + y + z')(x' + y' + z)(x' + y' + z')$$

Wszystkie przedstawione powyżej zapisy są równoważne. Jednakże, funkcje boolowskie często są stosowane do stworzenia realnych bramek logicznych. Aby zaoszczędzić pieniądze, należy sprawić, żeby te funkcje zajmowały jak najmniej bramek. Przykładowo, funkcja $f(x, y) = xy + x'y + x$ jest równoważna funkcji $f(x, y) = y + x$. Dlatego potrzeba metody, która pozwoli nam taką funkcję zminimalizować (zoptymalizować).

3.3 Metoda Karnaugh (ang. *Karnaugh map*)

Siatka Karnaugh jest równoważnym przedstawieniem funkcji boolowskiej. Zawiera wszystkie możliwe kombinacje wejść i każdej przyporządkowuje prawdę albo fałsz na wyjściu. Weźmy pewną funkcję `F(A,B,C)`. Ma ona trzy wejścia. Ze wzoru 2^n, siatka musi opisywać wartości ośmiu wariantów:

		BC			
		00	01	11	10
A	0	1	0	0	0
	1	0	1	0	0

W tym przypadku prawdę na wyjściu otrzymamy dla dwóch kombinacji wejściowych. Znajdźmy jedną z nich, przykładowo dla „jedynki" z dolnego rzędu. Odczytujemy zmienne z oznaczeń wierszy i kolumn. Wiersze definiują tutaj tylko wartość wejścia `A`, więc trywialne. Zapisujemy `A = 1`.

Nieco bardziej skomplikowane w tym przypadku są kolumny, wyznaczające jednocześnie wartości dwóch wejść. Hipotetycznie każda zmienna powinna być zapisana w osobnym wymiarze, jednak rozszerzyłoby to problem do zagadnienia 3D lub nawet 4D. Zamiast tego przy funkcjach trzech lub czterech zmiennych następuje łączenie wejść w pary, co znacząco upraszcza rysunki. Dlatego zapisujemy `BC = 01`, a następnie rozbijamy na `B = 0` oraz `C = 1`. Logiczną prawdę otrzymamy (nie wyłącznie!) dla `A = 1`, `B = 0`, `C = 1`.

A co w przypadku odwrotnym, gdyby szukać wartości wyjściowej dla podanych wejść? Weźmy nową siatkę, tym razem bardziej losowy układ i cztery zmienne.

AB \ CD	00	01	11	10
00	1	1	0	1
01	1	1	1	0
11	0	1	0	0
10	0	0	0	0

Chcemy znaleźć wartość dla `A = 0, B = 1, C = 1, D = 0`. W oznaczeniach wierszy jest `AB`, natomiast kolumny wyznaczają wartości wejść `CD`. Szukamy więc `AB = 01` oraz `CD = 10`.

AB \ CD	00	01	11	10
00	1	1	0	1
01	1	1	1	**0**
11	0	1	0	0
10	0	0	0	0

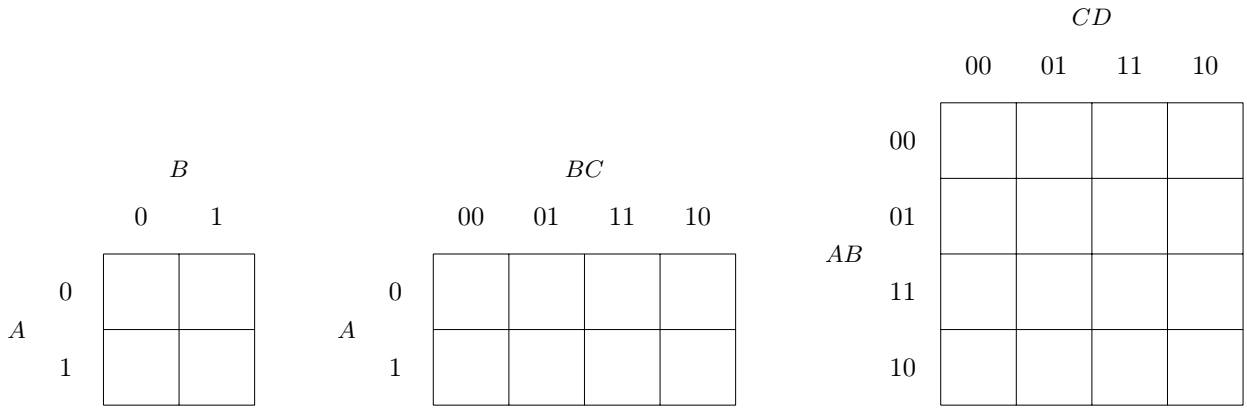
Dla podanego układu wejść nasza funkcja przyjmie wartość `0`, logiczny fałsz.

Jak widać, czytanie siatek Karnaugh jest bardzo proste. Większość zadań z tej kategorii dotyczy jednak znacznie trudniejszego zagadnienia, czyli samodzielnego rysowania siatki, wypełniania, a następnie minimalizacji. Dobra wiadomość: ogólny algorytm jest wspólny dla wszystkich odmian zadania. Różnica dotyczy jedynie sposobu wypełniania, ponieważ funkcja wejściowa może być przedstawiona na wiele równoważnych sposobów. Natomiast na wyjściu zawsze otrzymamy identyczny wynik, ale w zależności od treści zadania należy go odpowiednio zinterpretować.

3.3.1 Schematy siatek

Zacznijmy od podstaw. Tabelka Karnaugh ma boki będące potęgami liczby dwa, a ilość pól wynosi 2^n, gdzie jako n oznaczamy ilość zmiennych. Wiersze i kolumny (kolejność umowna) muszą zostać opisane kodem Gray'a i nazwane zmiennymi. Dla przypomnienia, kolejne wyrazy w tym kodzie różnią się dokładnie jednym bitem. Na potrzeby minimalizacji wykorzystamy kod trywialny (`0,1`) oraz kod dwubitowy (`00, 01, 11,`

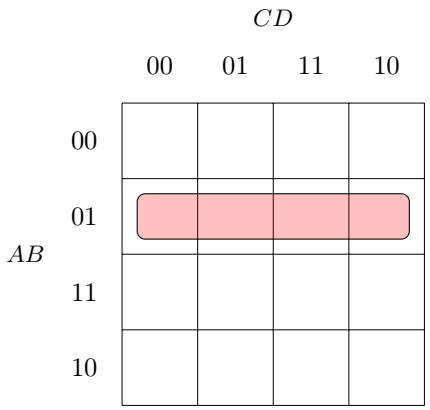
10). Natomiast zmienne dzielimy między „pion" i „poziom". Kod jednobitowy musi być powiązany z jedną zmienną, kod dwubitowy z dwiema zmiennymi. Doskonale widać to na poniższych tabelkach modelowych:



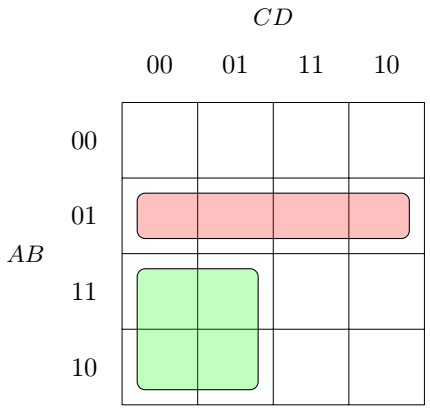
3.3.2 Wypełnianie siatki

W zadaniach można najczęściej natrafić na jeden z czterech podstawowych wariantów podania danych wejściowych - funkcja, opis, SOP, POS. Różnice między nimi dotyczą wyłącznie etapu wypełniania, później postępowanie znów będzie identyczne. Dla udowodnienia równoważności, wszystkie cztery przykłady zostały zrealizowane dla identycznej formuły.

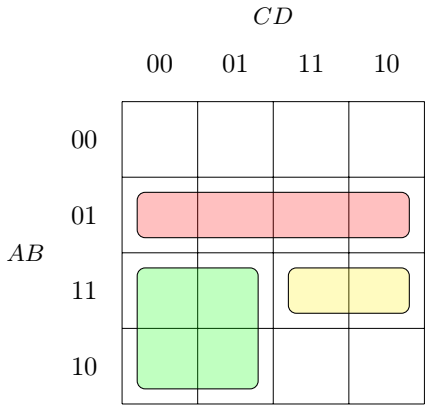
- **Funkcja boolowska**
  W pierwszej kolejności mnożymy wszystkie nawiasy i usuwamy składniki trywialne. Przykład po uporządkowaniu:
  `F(A,B,C,D) = A'B + A (BC + C') + DD' = A'B + AC' + ABC`
  Otrzymaliśmy tzw. sumę iloczynów (SOP). Z definicji logicznej alternatywy, jeśli którykolwiek iloczyn będzie miał wartość `1`, to cała funkcja przyjmie `1` na wyjściu. Wystarczy więc osobno zinterpretować wszystkie iloczyny. Przykładowo, `A'B` ma wartość `1` wtedy i tylko wtedy, gdy `A = 0` oraz `B = 1`, a pozostałe zmienne mogą być dowolne. Zgodnie z instrukcją podaną we wprowadzeniu, znajdźmy wszystkie komórki spełniające kryteria. Obie zmienne są na legendzie „pionowej", więc wybieramy po prostu cały wiersz `01`.



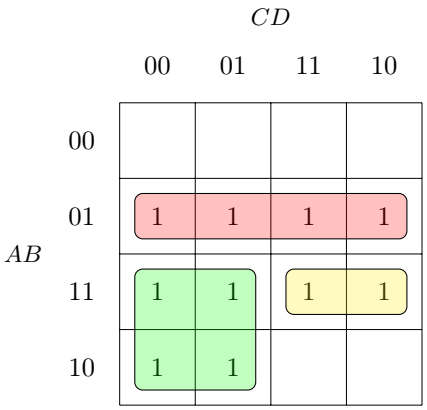
Następny składnik to `AC'`. Znajdujemy więc dwa wiersze z `A = 1` oraz dwie kolumny, gdzie `C = 0`, a następnie zaznaczamy część wspólną.



Ostatni iloczyn to `ABC`. Wybieramy jeden wiersz i dwie kolumny, częścią wspólną są dwie komórki.



W zaznaczone miejsca wstawiamy jedynki. Tabelkę bez zakreśleń przepisujemy, pozostałe miejsca uzupełniamy zerami. Z gotowym wynikiem przechodzimy do następnego kroku.



AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

- **notacja** Σ (skrócony zapis SOP)
  Pomimo pozornego skomplikowania, notacja skrócona Σ jest bardzo wygodna. Wykorzystuje ona fakt, że wiersze i kolumny są opisane kodem Gray'a, więc każdemu polu można przypisać unikatową liczbę binarną, łącząc oznaczenia wiersza i kolumny. Na początku najlepiej wykonywać trzy tabelki - pomocniczą binarną, pomocniczą dziesiętną i główną. Później można wykonywać zamianę w locie i zapisywać tylko wartości dziesiętne. Znając na pamięć kolejność komórek, nawet to staje się oczywiście niepotrzebne. W przykładach wykorzystamy natomiast metodę najdłuższą, ale najpewniejszą. Pierwsza z poniższych tabelek pomocniczych to złączone wartości wierszy i kolumn (w tej kolejności, ponieważ `AB` jest przed `CD`), a druga przedstawia zamianę wartości na system dziesiętny.

AB \ CD	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Nasz przykład to $F(A,B,C,D) = \Sigma m(4,5,6,7,8,9,12,13,14,15)$. Numery zakreślamy w pomocniczej tabelce dziesiętnej. Do głównej w analogiczne miejsca wpisujemy jedynki, uzupełniamy zerami.

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

- **Notacja $\prod$** (skrócony zapis POS)
  Wykorzystujemy identyczne tabelki pomocnicze jak dla notacji Σ, bardzo podobny jest również algorytm. Zaznaczamy w dziesiętnej tabelce podane numery. Jedyna różnica polega na tym, że w zaznaczone miejsca wstawiamy zera, a uzupełniamy jedynkami, czyli odwrotnie jak wcześniej. Teraz przykładem będzie funkcja $F(A,B,C,D) = M\prod(0,1,2,3,10,11)$. Zaznaczmy wszystkie sześć pól:

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

- **Polecenie opisowe**
  Może wydawać się trudne, jednak przy małej ilości zmiennych (do 5) wystarczające jest rozważenie przypadków w tradycyjnej tabelce.

  Zadanie 1. Funkcja zwracająca prawdę dla czterobitowych liczb większych od 3, których suma cyfr w zapisie dziesiętnym wynosi co najmniej 3.

  Zapiszmy wszystko, co może się przydać: cztery zmienne, wartość binarna, wartość dziesiętna, warunek 1 (oznaczony jako x), warunek 2 (oznaczony jako y) oraz końcową wartość, czyli koniunkcję warunków.

A	**B**	**C**	**D**	**(BIN)**	**(DEC)**	**x**	**y**	**x ∧ y**
0	0	0	0	0000	0	0	0	0
0	0	0	1	0001	1	0	0	0
0	0	1	0	0010	2	0	0	0
0	0	1	1	0011	3	0	1	0
0	1	0	0	0100	4	1	1	1
0	1	0	1	0101	5	1	1	1
0	1	1	0	0110	6	1	1	1
0	1	1	1	0111	7	1	1	1
1	0	0	0	1000	8	1	1	1
1	0	0	1	1001	9	1	1	1
1	0	1	0	1010	10	1	0	0
1	0	1	1	1011	11	1	0	0
1	1	0	0	1100	12	1	1	1
1	1	0	1	1101	13	1	1	1
1	1	1	0	1110	14	1	1	1
1	1	1	1	1111	15	1	1	1

Otrzymane wartości należy przepisać do siatki Karnaugh. Możemy to zrobić bezpośrednio z wartości A, B, C, D, natomiast w celu uniknięcia pomyłki zalecam zakreślanie jedynek w tabelce pomocniczej.

AB \ CD	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

3.3.3 Minimalizacja

Przykłady w poprzednich zadaniach (funkcja, Σ, $\prod$, opis) były równoważne, dzięki czemu algorytm minimalizacji jest wspólny dla wszystkich czterech podpunktów poprzedniego kroku. Dla przypomnienia, taką tabelkę wyznaczyliśmy z danych wejściowych:

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

Na wyjściu możemy otrzymać jedną z dwóch podstawowych postaci, SOP lub POS. Jeśli jest zaznaczone w poleceniu, wybieramy odpowiednią (lub obie). Jeśli brak informacji, domyślnie liczymy SOP.

- **Minimalny SOP**, suma iloczynów, np. `XY + Z`
  W celu wyznaczenia POS, zaznaczamy możliwie największe grupy jedynek o bokach 2^n. Dla funkcji czterech zmiennych dozwolone wartości to 1, 2 i 4. Zachodzenie grup na siebie jest wskazane, ponieważ im większa grupa, tym mniej symboli potrzeba do jej zapisu. W tym przypadku „zielony" prostokąt mógłby składać się tylko z dolnej części, ale każde rozszerzenie jest korzystne. Warto pamiętać o zasadzie, która pozwoli uniknąć błędów: niezależnie od wielkości siatki, każda kolejna zmienna w iloczynie powoduje zmniejszenie pola o połowę. Jedna opisuje obszar równy połowie pola, dwie to już 1/4, przy trzech zostaje 1/8, i tak dalej, a użycie wszystkich możliwych wskazuje na konkretne pole.

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

Tworzenie funkcji przebiega odwrotnie do wypełniania tabelki. Dla każdej grupy należy wypisać, jakie zmienne ją jednoznacznie definiują. W polu „czerwonym" `A, C, D` nie mają jednoznacznej wartości, więc zostają pominięte. Natomiast `B` na zakreślonym obszarze jest zawsze równe 1, a poza nim zawsze równe 0. Dlatego składnik odpowiadający największej grupie będzie równy po prostu `B`. Podobnie dla „zielonego" kwadratu. Łatwo można zauważyć, że obszar ten wyznaczają `A` oraz `C'`. Iloczyn wynosi `A'C`. Końcowy wynik to `SOP = AC' + B` .

- **Minimalny POS**, iloczyn sum, np. `(X + Y) (Z)`
  Analogiczne postępowanie przy POS. Tabelkę ponownie przepisać, ale tym razem zakreślić zera. Ten konkretny przykład pokazuje ważną cechę siatki Karnaugh - formalnie jest ona torusem[1], więc naprzeciwległe krawędzie są również połączone ze sobą, co należy uwzględniać przy szukaniu grup maksymalnych. Opisana sytuacja występuje w przypadku grupy „zielonej", utworzonej ze skrajnych rzędów.

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

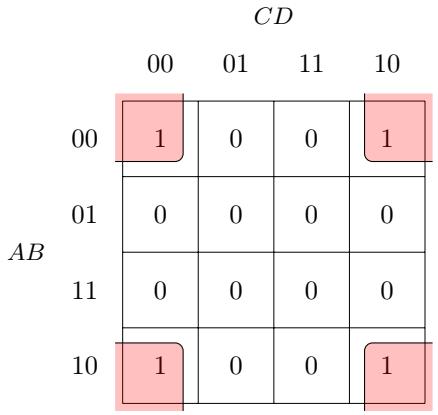
AB \ CD	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	0	0

Z tabelki jednoznacznie wynika, że tym razem „zielona" grupa jest definiowana przez `C` oraz `B'`, natomiast „czerwona" przez `A'` i `B'`. Tutaj dochodzi dodatkowy ważny krok! Przed połączeniem w sumy, należy zanegować każdą zmienną. Ostateczny wynik to `SOP = (C'+B)(A+B)`.

Wykorzystany przykład jest na tyle mało zaawansowany, że można szybko wykazać równoważność obu postaci. Najprościej przemnożyć składniki minimalnego POS: `(C' + B)(A + B) = AC' + B`. Wynik identyczny z minimalnym SOP. Przy skomplikowanych funkcjach postać nadal będzie równoważna, ale jest to znacznie trudniejsze to zaobserwowania, ponieważ mnożąc POS możemy otrzymać SOP dowolny, nie minimalny.

Uwaga! Czasami (chociaż rzadko) zdarza się, że jedynki/zera wystąpią na wszystkich czterech rogach. Wtedy należy zgrupować je w kwadrat, korzystając jednocześnie z zasad łączenia przeciwległych boków w pionie oraz w poziomie. Dla siatki czterech zmiennych jak poniżej, taki iloczyn zostanie oznaczony `B'D'`.

[1] model 3D można zakupić w Biedronce pod nazwą handlową donut



3.3.4 Interpretacja wyniku

Z poprzedniego podpunktu otrzymaliśmy POS i/lub SOP. Przyjmując że poprawnie wybraliśmy możliwie najmniejszą ilość maksymalnych grup, są to oczywiście postaci minimalne. Jeśli w treści zadania jest wymagana postać kanoniczna, dodatkowo sztucznie uzupełniamy wszystkie brakujące zmienne. Do SOP bez zmiany wartości można dodać `A' A`. Odpowiednikiem dla POS jest mnożenie `(A + A')`. Naszym przykładem była funkcja `F(A,B,C,D)`, ale wszystkie składniki zawierające `D` były trywialne, więc zmienna ta została wykluczona już na początku. Dodajmy ją w celu uzyskania postaci kanonicznej. POS kanoniczny to `AC' + B + DD'`, natomiast kanoniczny POS przybierze postać `(C' + B)(A + B)(D + D')`. Czasami elementem zadania może być też narysowanie odpowiedniej kombinacji bramek logicznych. Wtedy wykorzystujemy minimalny SOP i postępujemy jak przy projektowaniu układów logicznych z funkcji boolowskiej (w osobnym rozdziale).

3.3.5 „Dla tych, którzy chcą wiedzieć więcej..."

- **Stany nieoznaczone**
  Czasami w zadaniu wskazane są kombinacje, które nie muszą przyjmować żadnej określonej wartości. Najczęściej dotyczy to postaci skróconej SOP lub POS (notacje Σ i $\prod$), gdzie pola te oznaczone są literą `d`. Ogólna zasada jest bardzo prosta - stan nieokreślony traktujemy jako zero lub jedynkę, w zależności od tego, co pozwoli lepiej zminimalizować funkcję. Przykład: $F(A,B,C,D) = \Sigma m(0,1,4) + d(5,7)$

AB \ CD	00	01	11	10
00	1	1	0	0
01	1	X	X	0
11	0	0	0	0
10	0	0	0	0

AB \ CD	00	01	11	10
00	1	1	0	0
01	1	X	X	0
11	0	0	0	0
10	0	0	0	0

Pierwszy stan nieokreślony potraktowaliśmy jako zero, drugi jako jedynkę, dzięki czemu udało się utworzyć pojedynczą grupę 2x2 zamiast dwóch 1x2.

- **Inne zestawy bramek**
  Standardowe zadania z SOP, POS i minimalizacji zezwalają tylko na negację, alternatywę oraz koniunkcję, natomiast przy projektowaniu prawdziwych układów do dyspozycji są również pozostałe bramki logiczne. Wykorzystując `XOR` lub `XNOR`, możliwa jest tzw. minimalizacja skośna.

A \ B	0	1
0	0	1
1	1	0

A \ B	0	1
0	1	0
1	0	1

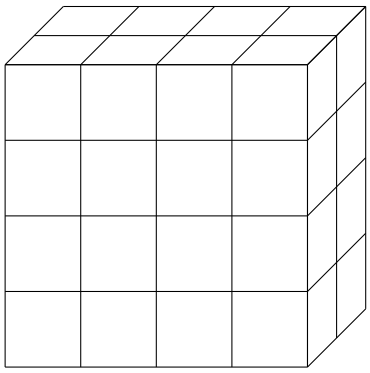
Pierwsza tabelka to `A'B + AB'`, co jest dosłowną definicją `A XOR B`. Natomiast drugą można zapisać jako `A'B' + BA`, czyli `A XNOR B`. Zamiast pięciu bramek jedna! A jak to wygląda po przeliczeniu na podstawowe elementy? Dwie negacje, dwie koniunkcje i alternatywa wymagają aż 22 tranzystorów [2]. Natomiast `XNOR` to 8 tranzystorów, `XOR` zaledwie 6. Zysk jest oczywisty. Podobne rozumowanie można przeprowadzić dla większych tabel w połączeniu z minimalizacją klasyczną, stosując łączenie po skosie. Z przyczyn osobistych, pozostawiam to Czytelnikowi jako ćwiczenie.

- **Szybkie zamiany**
  Jeśli otrzymaliśmy SOP lub POS w postaci skróconej i chcemy jedynie odtworzyć funkcję bez minimalizacji, można to zrobić nie wykorzystując tabelek. Każdą liczbę z nawiasu zapisujemy w postaci binarnej na tylu bitach, ile funkcja ma wejść. Dla $F(A,B,C) = \Sigma m(1,2,7)$ będą to 001, 010, 111. Kolejne bity zamieniamy na kolejne litery. 1 oznacza daną zmienną, 0 jej negację. $001 \rightarrow A'B'C$, $010 \rightarrow A'BC'$, $111 \rightarrow ABC$. Końcowym wynikiem jest $F(A,B,C) = A'B'C + A'BC' + ABC$. Analogicznie $F(X,Y,Z) = \prod M(0,3,6)$, ale dla POS zasada jest odwrotna: 1 to negacja, 0 niezmieniona wartość. Otrzymujemy 000, 011, 110. Z tego $F(X,Y,Z) = (A+B+C)(A+B'+C')(A'+B'+C)$

- **Więcej zmiennych!**
  Powyżej czterech wejść, teoretycznie można łączyć zmienne trójkami, ale jest to znacznie mniej intuicyjne niż w przypadku par. Przykładowo suma iloczynów `CD' + CD` to oczywiście prosta formuła `C`. Jednak na pojedynczej siatce pięciu zmiennych mogłaby ona być widoczna jako dwie oddzielne grupy, których nie da się połączyć żadną z wcześniejszych zasad.

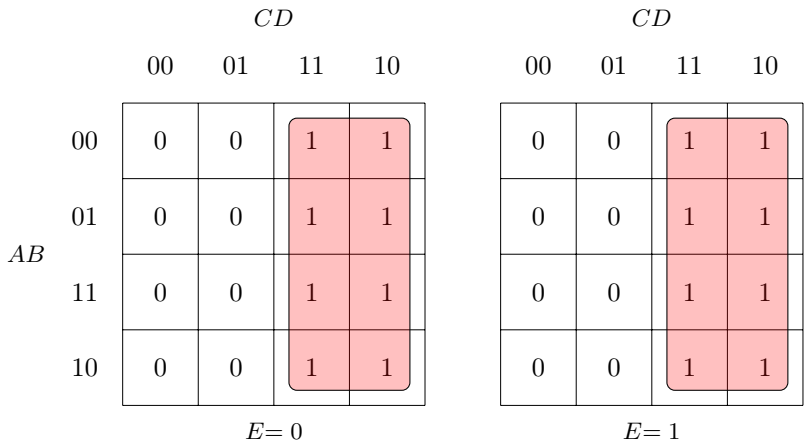
DE \ *ABC*	000	001	011	010	110	111	101	100
00	0	1	1	0	0	1	1	0
01	0	1	1	0	0	1	1	0
11	0	1	1	0	0	1	1	0
10	0	1	1	0	0	1	1	0

W takiej sytuacji należałoby omówić reguły łączenia symetrycznego. Prościej jest jednak wykorzystać to, czego cały czas unikaliśmy, czyli kolejny wymiar. Wystarczy wyobrazić sobie połówkę sześcianu, składającą się z 32 małych kostek. Każdej kostce możemy oczywiście przypisać prawdę lub fałsz.



Na potrzeby minimalizacji wystarczą nam przekroje, dwie osobne siatki Karnaugh[2]. Dalej postępujemy identycznie jak przy standardowej minimalizacji, pamiętając oczywiście że analogiczne pola obu siatek są również złączone ze sobą. Poniżej nasz wcześniejszy przykład `CD' + CD`. Tym razem bezproblemowo otrzymaliśmy postać minimalną `F(A,B,C,D,E) = C`.

[2]W podstawowym algorytmie minimalizacji zostało zaznaczone, że każda siatka Karnaugh jest torusem. To prawda. Dobra rada: nie myśl teraz o tym. Cztery wymiary są fajne, ale bezpieczniej jest zostawić je komuś z 214b/D1.



- **A może dwa wyjścia?**
  W przypadku projektowania układu posiadającego więcej niż jedno wyjście, zazwyczaj zadowalające efekty daje zapisanie dla każdego z nich osobnej funkcji boolowskiej, a następnie oddzielna minimalizacja. Warto również zaznaczyć powtarzające się iloczyny (w SOP), dzięki czemu będzie możliwe wykorzystanie danego zestawu bramek wielokrotnie. Natomiast wyznaczenie w takim przypadku postaci jednoznacznie minimalnej jest problemem NP-trudnym, znacząco wykraczającym poza zakres tego kompendium. W przypadku zainteresowania tematem, odsyłam do literatury [1].

Rozdział 4

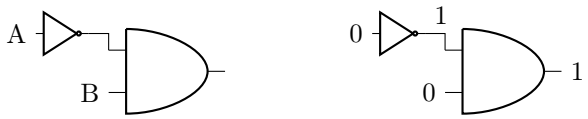
Układy kombinacyjne

> Uczeni wyliczyli, że jest tylko jedna szansa na bilion, by zaistniało coś tak całkowicie absurdalnego. Magowie obliczyli, że szanse jedna na bilion sprawdzają się w dziewięciu przypadkach na dziesięć.
>
> (Terry Pratchett)

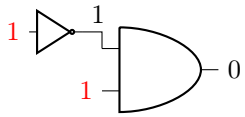
4.1 Hazard

4.1.1 Trywialny przykład

Jako hazard określamy błędne stany wyjść układów cyfrowych, które powstają podczas przełączania poszczególnych bramek. Poniższy rysunek przestawia bardzo prosty™ układ logiczny. Po lewej stronie bramka `NOT` neguje wartość zmiennej `A`, natomiast bramka `AND` wykonuje koniunkcję wartości `A'` oraz `B`. Oba wejścia mają początkowo wartość 0.

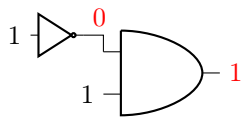


Sprawdźmy teraz, co się stanie, gdy równocześnie zmienimy wartość obu zmiennych na 1. Operujemy na schemacie teoretycznym, dlatego właściwości samych połączeń pomijamy. Natomiast istotne opóźnienie wprowadzają bramki logiczne. Dla uproszczenia przyjmijmy, że każda bramka zwraca na wyjściu prawidłową wartość dopiero po (identycznym) czasie t.

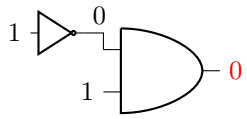


W pierwszym kroku ($T = 0$) zmieniły się wyłącznie wartości na wejściach, wartości wyjściowe pozostały bez zmian. Sprawdźmy, jak zachowują się teraz poszczególne elementy. Bramka `NOT` otrzymuje wartość 1 na

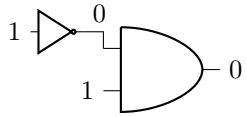
wejściu, ale z uwagi na opóźnienie stan wyjściowy jest nadal równy 1![1] Podobnie bramka `AND` ma oba wejścia w stanie 1, natomiast na wyjściu nadal 0. Zwiększmy czas T o pojedyncze opóźnienie, czyli t.



Bramka `NOT` otrzymała wcześniej wartość 1, więc prawidłowo zwraca teraz 0. Analogicznie bramka `AND` - wejścia były ustawione na 1, zgodnie z definicją taka wartość pojawiła się też na wyjściu. Końcowa wartość w układzie to aktualnie 1. Jednak to jeszcze nie koniec, ponownie zwiększamy czas o wartość t.



Żadne zmiany nie nastąpiły przy bramce `NOT`, która już wcześniej się ustabilizowała. Natomiast jedno z wejść `AND` w poprzednim kroku osiągnęło wartość 0, więc bramka ponownie musi przyjąć stan 0. Całość została doprowadzona do stanu stabilnego, dalsze zwiększanie czasu nie spowoduje zmian.



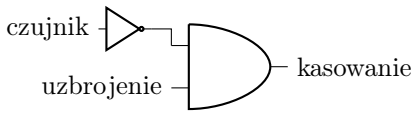
Nasz prosty układ jest równoważny funkcji boolowskiej `F(A,B) = A'B`. Przyjmuje ona logiczną prawdę wtedy i tylko wtedy, gdy `A = 0` oraz `B = 1`. Jednak przez chwilę otrzymaliśmy na wyjściu wartość 1, chociaż oba wejścia miały identyczną. Jest to modelowy przykład hazardu. Z powodu niedoskonałości elementów elektronicznych, podczas zmiany wartości wejściowych przez krótki czas wyjście może mieć stan inny od oczekiwanego.

4.1.2 Złe, bardzo złe i tragiczne skutki

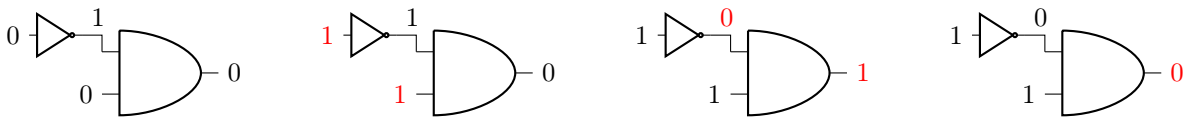
Negatywne efekty hazardu mogą być mało zauważalne (ale nadal niepożądane), gdy dotyczą wyłącznie komunikacji z użytkownikiem, np. przez wyświetlacz numeryczny. Doprowadzenie do stanu stabilnego z punktu widzenia człowieka jest natychmiastowe, ponieważ odświeżanie trwa krócej niż czas reakcji. Nieporównywalnie gorszy jest hazard w układach, które wykonują dalsze działania lub zapisują dane. Wtedy pojedynczy błąd może być propagowany na cały układ lub nawet rejestrowany w pamięci. Najbardziej niepożądany wariant występuje w przypadku, gdy skutki wykonania danej operacji są nieodwracalne.

Rozważmy to na przykładzie. Mamy centralkę alarmową, która wysyła dwa sygnały. Jeden wskazuje na uzbrojenie systemu (0 - nieaktywny, 1 - aktywny), drugie to stan czujnika ruchu (1 - brak problemów, 0 - zagrożenie). Chcemy stworzyć prosty układ, który uruchomi automat nadpisujący, jeśli w stanie „uzbrojnym" czujnik wykryje włamanie. Automat to zewnętrzne urządzenie, aktywowane impulsem 1.

[1] Eksperci go nienawidzą, odkrył jeden prosty sposób, jak uniknąć żartów o silni.



Przykładową realizacją jest układ z poprzedniego podpunktu. Pomimo wykazanego problemu z hazardem, może się wydawać że całość zadziała poprawnie, ponieważ wejścia są aktywowane niezależnymi czynnikami. Załóżmy jednak, że podczas standardowego działania alarmu dopływ energii został odcięty, przez co na wejściach naszego okładu zanika sygnał. Gdy wróci zasilanie, centralka alarmowa zacznie znowu wysyłać oba sygnały - uzbrojenie systemu oraz stan czujnika ruchu. Hipotetycznie możliwa jest więc sytuacja podobna do opisanej wyżej - jednoczesne przełączenie obu wejść. Przypomnijmy, co się wtedy dzieje:

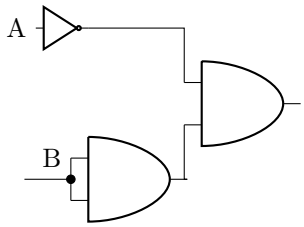


Po trzecim kroku do automatu został wysłany impuls aktywujący, mimo że nie nastąpiło włamanie. Wszystkie dane zostały niepotrzebnie zniszczone.

Oczywiście całość jest tylko absurdalnym, teoretycznym rozważaniem. Przełączenie obu wejść w identycznym czasie jest niemalże nieprawdopodobne. Dobrze jednak pokazuje to mechanizm hazardu, który (najczęściej w mniej spektakularnej formie) może wystąpić również przy całkowicie realnych zastosowaniach.

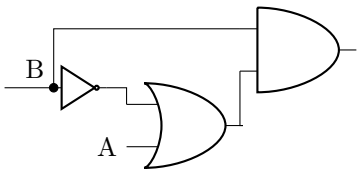
4.1.3 Zapobieganie

Paradoksalnie, dla tak prostej funkcji jak powyżej, całkowite wyeliminowanie hazardu jest niemożliwe. Do uzyskania poprawnego wyniku musimy zawsze zanegować sygnał A, natomiast B pozostawić niezanegowane, przez co czas propagacji sygnału będzie różny. Czysto teoretycznym rozwiązaniem byłoby zrównoważenie jednej bramki `NOT` odpowiednim układem opóźniającym. Niemożliwe jest jednak zbudowanie z samych bramek `NOT` układu, który bez negacji opóźni sygnał o tyle, co pojedyncza bramka `NOT`. A gdyby zamiast tego wykorzystać bramkę `AND`?

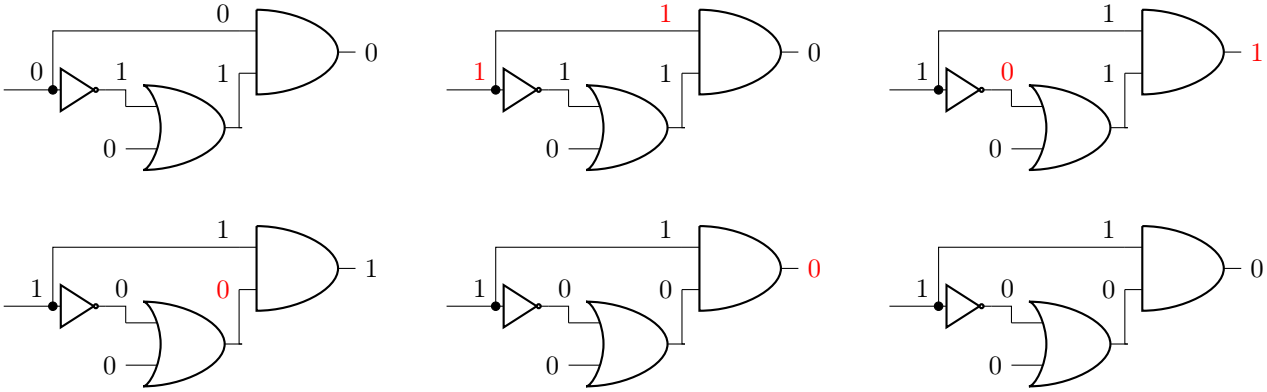


Pod względem teoretycznym wszystko się zgadza i tak też wskaże większość symulatorów obwodów. Niestety, w praktyce bramki mają różne opóźnienia. Przy samym stwierdzaniu zjawiska hazardu mogliśmy pozwolić sobie na zaokrąglanie tego do stałej wartości, ponieważ kluczowy był sam fakt występowania zjawiska, a nie konkretne stany pośrednie. Natomiast chcąc zrównoważyć czasy propagacji sygnału w rzeczywistym układzie, musielibyśmy uwzględniać odmienne charakterystyki.

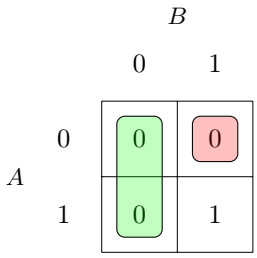
Wykazaliśmy, że w omawianym przypadku nie da się w żaden klasyczny sposób usunąć hazardu. Powrócimy do tego układu przy układach synchronicznych, a teraz spróbujemy od nowa na przykładzie modelowym. Nowa funkcja to `B (A + B')`.



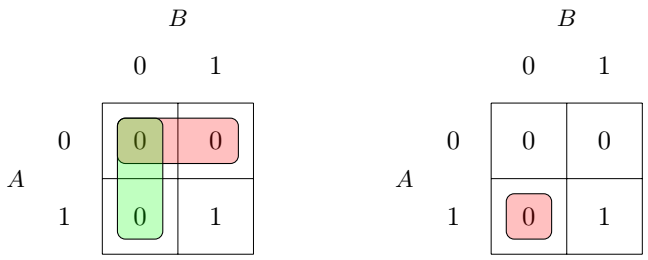
Początkowo `A = 0, B = 0`. Sprawdźmy, jak się zachowa funkcja po zmianie stanu B na 1.



I znowu to samo. W jednym z kroków na wyjściu otrzymaliśmy wartość 1, chociaż poprawna wartość to zero. Istnieje jednak zasada, która pozwoli nam pozbyć się problemu. Hazard występuje, gdy na siatce Karnaugh sąsiednie grupy nie są ze sobą połączone. Sprawdźmy więc nasz układ.

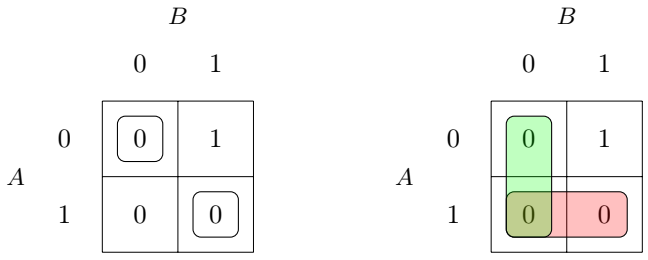


Analizowaliśmy zmianę `B = 0` na `B = 1` przy `A = 0`, dokładnie w tym miejscu występuje przerwa. Wyeliminowanie hazardu jest tutaj bardzo proste, wystarczy rozszerzyć jedną z grup albo policzyć SOP, gdzie jest tylko jedna grupa jednoelementowa.



W obu przypadkach nowa funkcja to `AB`, pojedyncza bramka logiczna bez negacji. Z definicji nie może pojawić się hazard. Jednocześnie otrzymaliśmy postać minimalną, co jest raczej rzadkim zjawiskiem. Zazwyczaj eliminacja podobnych problemów powoduje zwiększenie ilości bramek i zwiększenie stopnia skomplikowania.

Dlaczego nie dało się w żaden sposób uniknąć hazardu przy poprzednim przykładzie? Wykazaliśmy to już analizując układ, czas na wyjaśnienie teoretyczne. Była to w pewnym sensie, forma oszustwa. Rozważaliśmy jednoczesną zmianę dwóch zmiennych, na co większość prostych układów nie jest odporna. Sprawdźmy tabelkę. Rozważaliśmy przejście między zerami, narysujmy grupy jak dla POS.

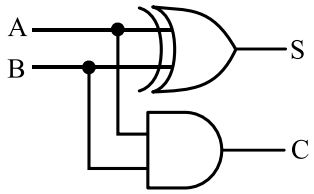


Na pierwszej tabelce jest zaznaczone, między którymi polami chcieliśmy przeskoczyć. Stykają się one tylko rogami. Druga siatka pokazuje, że nie potrafimy zawrzeć punktu styku w żadnej grupie.

4.2 Sumatory

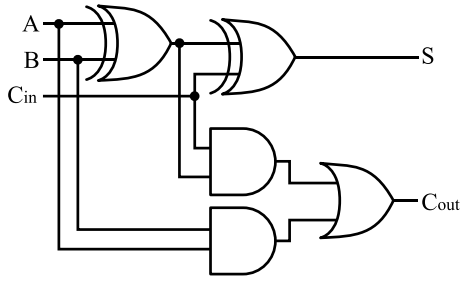
Półsumator Półsumator to prosty układ logiczny, realizujący dodawanie pojedynczych bitów. Zasada działania jest identyczna jak przy dodawaniu pisemnym. Jeśli wynik dodawania nie mieści się na pojedynczym bicie (1+1), na wyjściu sumy zwracane jest zero, a ustawiany jest bit przeniesienia.

a	**b**	**S**	**C**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Sumator pełny Działa identycznie jak półsumator, ale uwzględnia przeniesienie z poprzedniego sumatora, dzięki czemu można je łączyć kaskadowo i dodawać liczby wielobitowe.

a	**b**	**c**	**S**	**C**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



4.3 Komparatory

Komparator jest jednym z najprostszych (pod względem logicznym) układów. Zwraca prawdę wtedy i tylko wtedy, gdy dane wejścia są ze sobą w określonej relacji. Najprostszym komparatorem jest bramka `XNOR`, która zwraca prawdę tylko w przypadku równości.

Rozdział 5

Architektura procesora

W matematyce nie ma nic do zrozumienia, trzeba się po prostu przyzwyczajać.

(John von Neumann)

5.1 Architektura ze względu na przechowywanie danych i rozkazów

5.1.1 Architektura von Neumanna

Pierwsza architektura komputera w historii, z roku 1945. Cecha charakterystyczna to przechowywanie danych wspólnie z instrukcjami oraz kodowanie ich w identyczny sposób. Przeciwieństwo architektury harwardzkiej. System w architekturze von Neumanna powinien mieć skończoną i funkcjonalnie pełną listę rozkazów.

Podstawowe komponenty:

- pamięć (przechowuje dane oraz instrukcje)
- jednostka sterująca (pobiera dane, przetwarza instrukcje)
- jednostka arytmetyczno-logiczna (wykonuje operacje arytmetyczne, komunikuje się z wejściem/wyjściem)
- urządzenia wejścia/wyjścia (pozwalają na interakcję z użytkownikiem)

5.1.2 Architektura harwardzka

W połowie XX wieku podstawowa architektura komputerowa, obecnie jest wykorzystywana głównie w mikrokontrolerach. Kluczową różnicą względem architektury von Neumanna jest osobna pamięć na instrukcje.

5.2 Architektura ze względu na złożoność logiki rozkazów

5.2.1 Architektura CISC

Wiele złożonych instrukcji, skomplikowany dekoder rozkazów , bezpośrednie operacje na pamięci.

5.2.2 Architektura RISC

Uproszczony zestaw instrukcji, ograniczenie komunikacji z pamięcią, duża liczba rejestrów, jeden tryb adresowania.

5.3 Architektura ze względu na przetwarzanie wektorów danych

5.3.1 Architektura SIMD

Różne strumienie danych są przetwarzanych w oparciu o pojedynczy strumień rozkazów. Współczesne procesory posiadają zestawy instrukcji zgodne z tą ideą (MMX, SSE, itd.), co pozwala znacząco przyspieszyć działania powtarzalne (np. obróbkę dźwięku i obrazu)

5.3.2 Architektura MIMD

Wiele procesorów pracujących niezależnie, równoległe przetwarzanie na poziomie danych i rozkazów.

Rozdział 6

Pamięć komputera

()

6.1 Pamięci ulotne

6.1.1 SRAM

6.1.2 DRAM

6.1.3 DDR

6.2 Pamięci trwałe

6.2.1 Pamięć magnetyczna

6.2.2 Flash

6.2.3 ROM

6.2.4 Pamięć optyczna

CD

DVD

Bluray

Rozdział 7

Wybrane techniki zarządzania pamięcią procesora

7.1 Metody ochrony pamięci

7.1.1 Segmentacja pamięci

7.1.2 Stronnicowanie pamięci

7.2 Podział pamięci

7.2.1 Wyrównywanie (ang. *alignment*) na poziomie procesora

7.3 Pamięć skojarzeniowa procesora

Rozdział 8

Procesor 6502

To jest wszystko bardzo proste

(turbo debadżer)

Procesor 6502, a właściwie mikroprocesor, był przełomowym osiągnięciem techniki. Jedną z największych jego zalet była bardzo niska cena, co zaowocowało wzrostem ilości komputerów domowych. Niezwykle prosty oraz efektywny design był inspiracją dla projektantów procesorów ARM.

8.1 Rejestry

Z punktu widzenia wybitnego programisty assemblera rejestry 6502 nie są zbyt interesujące. Bezpośredni dostęp posiada on jedynie do trzech rejestrów nazywanych:

- **A** - Akumulator - 8 bit
- **X** - 8 bit
- **Y** - 8 bit

Dodatkowo występują :

- **PC** - Licznik programu - 16 bit
- **SP** - Stack pointer - 8 bit
- **P** - Rejestr flag procesora - 8 bit (bit 5 nie jest używany, powinien zawsze być jedynką)

8.2 Flagi procesora

6520 posiada następujące flagi:

- **Z** - Flaga zero - Zapalana gdy operacja arytmetyczna, bądź logiczna dała wynik 0

- **C** - Flaga przeniesienia - Zapalana gdy przy wykonywaniu operacji dodawania zaistniało przeniesienie z najbardziej znaczącego bitu oraz odejmowania jeśli wymagane jest zapożyczenie. Flaga jest również czyszczony przy odejmowaniu jeśli zapożyczenie nie jest wymagane
- **V** - Flaga przepełnienia - Zapalana gdy operacja arytmetyczna zwraca wynik przekraczający swoim rozmiarem jeden bajt
- **N** - Flaga znaku - Zapalana jeśli wynikiem operacji jest liczba negatywna, czyszczona w przeciwnym wypadku
- **D** - Flaga trybu dziesiętnego - Zmienia zachowanie operacji dodawania i odejmowania z przeniesieniem na tryb BCD
- **B** - Flaga wstrzymania - Zapalona gdy przerwanie systemowe zostało wywołane
- **I** - Flaga przerwania - Zapalona umożliwia na używanie przerwań

8.3 Tryby adresowania

Tryb adresowania to sposób w jaki procesor uzyskuje dane potrzebne do wykonania aktualnej instrukcji. W przypadku 6502 trybów jest około trzynastu. Poniższa lista zawiera tylko najważniejsze:

- Akumulator - @ - Podana instrukcja odnosi się do danych zawartych w akumulatorze
- Immediate - #$1234 - Dane do instrukcji znajdują się bezpośrednio po opcodzie (np. wartość liczbowa)
- Implied - Adresowanie jest pomijane jako, że wynika bezpośrednio z instrukcji
- Relative - Używany przy rozgałęzieniach. Bajt po opcodzie zawiera długość ewentualnego skoku (-128 do 127 bajtów)
- Absolute - $1234 - Dwa bajty po opcodzie zawierają dokładny adres w pamięci
- Zero-page - $12 - Podobne do Absolute ale używane do adresowania tylko pierwszej strony w pamięci
- Indirect - ($1234) - Podobnie do Absolute ale adres skoku zawarty jest w pamięci, a nie w dwóch bajtach po opcodzie (Używany tylko przez instrukcję JMP)
- Absolute Indexed - $1234,X $1234,Y - Rozszerzenie trybu Absolute o możliwość dodania do adresu zawartości rejestrów X oraz Y
- Zero-page Indexed - $12,X $12,Y - Podobne do Absolute Indexed ale używane do adresowania tylko pierwszej strony w pamięci
- Indirect Indexed - ($12),Y - Rozszerzenie trybu Indirect o możliwość dodania do adresu z pamięci zawartości rejestru Y
- Indexed Indirect - ($12,X) - Rozszerzenie trybu Indirect o możliwość dodania do adresu zawartości rejestru X, a potem pobranie z nowo uzyskanego adresu zawartości

W przypadku adresowania typu absolute na adres przeznaczone są dwa bajty. Oznacza to 16 bitów na których można zapisać adres. Pozwala to na odwołania do 65536 (2^{16}) komórek w pamięci.

8.3.1 Etykiety

Etykiety no nic innego jak sposób aliasowania konkretnych miejsc w kodzie naszego programu. Niwelują one problem wynikający z tego, że np. po dodaniu jednej linii kodu, wszystkie linie poniżej zmieniają swój adres i trzeba by było zmieniać wartości wszystkich skoków absolutnych. Ale nie trzeba! Bo mamy cudowne etykiety i dzięki temu pisanie staje się bardzo proste

8.4 Stos

Po przestrzeni zarezerwowanej jako zero-page w 6502 następuje stos. Rozpoczyna się on na adresie 0x0100 oraz kończy na 0x01FF. Mamy do dyspozycji następujące operacja na stosie:

- PHA - Dodaje wartość na szczyt stosu
- PHP - Interpretowany języ... Umieszcza na stosie rejestr flag (Analogiczną, lecz bezpieczniejszą jest instrukcja ADA)
- PLA - Ściąga wartość ze szczytu stosu i umieszcza ją w rejestrze A
- PLP - Ściąga wartość ze szczytu stosu i umieszcza ją w rejestrze flag

8.5 Wybrane rozkazy procesora

- LDA - Ładuje dane do akumulatora. Wpływa na flagi N oraz Z
- STA - Zapisuje zawartość akumulatora w pamięci. Wpływa na flagi N oraz Z
- LDX - Ładuje dane do rejestru X. Wpływa na flagi N oraz Z
- STX - Zapisuje zawartość rejestru X w pamięci. Wpływa na flagi N oraz Z
- LDY - Ładuje dane do rejestru Y. Wpływa na flagi N oraz Z
- STY - Zapisuje zawartość rejestru Y w pamięci. Wpływa na flagi N oraz Z
- ADC - Dodaje zawartość akumulatora do wartości w pamięci z przeniesieniem. Wynik zapisuje w akumulatorze. Wpływa na flagi N, V, Z oraz C
- INC - Zwiększa zawartość pamięci o 1. Wpływa na flagi N oraz Z
- SBC - Odejmuje zawartość akumulatora od wartości w pamięci z przeniesieniem. Wynik zapisuje w akumulatorze. Wpływa na flagi N, V, Z oraz C
- DEC - Zmniejsza zawartość w pamięci o 1. Wpływa na flagi N oraz Z
- CMP - Porównuje zawartość akumulatora z wartością w pamięci. Wpływa na flagi N, Z oraz C
- CPX - Porównuje zawartość rejestru X z wartością w pamięci. Wpływa na flagi N, Z oraz C
- CPY - Porównuje zawartość rejestru Y z wartością w pamięci. Wpływa na flagi N, Z oraz C

- AND - Wykonuje bitową operację AND na akumulatorze oraz pamięci. Wynik zapisuje w akumulatorze. Wpływa na flagi N oraz Z
- ORA - Wykonuje bitową operację OR na akumulatorze oraz pamięci. Wynik zapisuje w akumulatorze. Wpływa na flagi N oraz Z
- EOR - Wykonuje bitową operację OR na akumulatorze oraz pamięci. Wynik zapisuje w akumulatorze. Wpływa na flagi N oraz Z

Tablica 8.1: Dostępne tryby adresowania podstawowych instrukcji 6502

OC	**#$12**	**$1234**	**$12**	**($1234)**	**$1234,X**	**$1234,Y**	**$12,X**	**$12,Y**	**($12),Y**	**($12,X)**
LDA	✓	✓	✓	-	✓	✓	✓	-	✓	✓
STA	-	✓	✓	-	✓	✓	✓	-	✓	✓
LDX	✓	✓	✓	-	-	✓	-	✓	-	-
STX	-	✓	✓	-	-	-	-	✓	-	-
LDY	✓	✓	✓	-	✓	-	✓	-	-	-
STY	-	✓	✓	-	-	-	✓	-	-	-
ADC	✓	✓	✓	-	✓	✓	✓	-	✓	✓
INC	-	✓	✓	-	✓	-	✓	-	-	-
SBC	✓	✓	✓	-	✓	✓	✓	-	✓	✓
DEC	-	✓	✓	-	✓	-	✓	-	-	-
CMP	✓	✓	✓	-	✓	✓	✓	-	✓	✓
CPX	✓	✓	✓	-	-	-	-	-	-	-
CPY	✓	✓	✓	-	-	-	-	-	-	-
AND	✓	✓	✓	-	✓	✓	✓	-	✓	✓
ORA	✓	✓	✓	-	✓	✓	✓	-	✓	✓
EOR	✓	✓	✓	-	✓	✓	✓	-	✓	✓

- ASL - Przesunięcie bitowe w lewo. Wpływa na flagi N, Z oraz C
- ROL - Przesunięcie bitowe w lewo z przeniesieniem. Wpływa na flagi N, Z oraz C
- LSR - Przesunięcie bitowe w prawo. Wpływa na flagi N, Z oraz C
- ROR - Przesunięcie bitowe w prawo z przeniesieniem. Wpływa na flagi N, Z oraz C

- TXA - Przesyła wartość z rejestru X do A
- TYA - Przesyła wartość z rejestru Y do A
- TAX - Przesyła wartość z rejestru A do X
- TAY - Przesyła wartość z rejestru A do Y
- TXS - Przesyła wartość z rejestru X do SP

Tablica 8.2: Dostępne tryby adresowania podstawowych przesunięć bitowych 6502

OC	**@**	**#$12**	**$1234**	**$12**	**$1234,X**	**$1234,Y**	**$12,X**
ASL	✓	-	✓	✓	✓	-	✓
ROL	✓	-	✓	✓	✓	-	✓
LSR	✓	-	✓	✓	✓	-	✓
ROR	✓	-	✓	✓	✓	-	✓

Tablica 8.3: Lista skoków warunkowych (a) oraz instrukcje flag (b) w 6502

(a) Skoki

OC	**Warunek**
BNE	Z = 0
BEQ	Z = 1
BCC	C = 0
BCS	C = 1
BPL	N = 0
BMI	N = 1
BVC	V = 0
BVS	V = 1

(b) Instrukcje

OC	**Działanie**
CLC	Z = 0
SEC	Z = 1
CLI	I = 0
SEI	I = 1
CLV	V = 1
CLD	D = 0
SED	D = 1

- TSX - Przesyła wartość z rejestru SP do X
- INX - Zwiększa wartość rejestru X o 1
- INY - Zwiększa wartość rejestru Y o 1
- DEX - Zmniejsza wartość rejestru X o 1
- DEY - Zmniejsza wartość rejestru Y o 1
- JMP - Skacze to podanego miejsca w pamięci. Umożliwia adresowanie Absolute oraz Indirect
- JSR - Skacze do podprogramu. Adresowanie wyłącznie Absolute
- RTS - Wraca z podprogramu
- NOP - Nie robi nic jak polski robotnik na budowie

8.6 Przykładowe programy

Konwersja do BCD:

```
bcdconv         lda #0
                sta res
```

```
                sta res+1
                ldx #8
                sed
loop            asl val
                lda res
                adc res
                sta res
                lda res+1
                adc res+1
                sta res+1
                dex
                bne loop
                cld
                rts

val             dta b(%01101101)

res             dta b(0),b(0)
```

Konwersja do tekstu szesnastkowego:

```
hex             pha
                jsr digit
                pla
                lsr @
                lsr @
                lsr @
                lsr @
digit           and #%00001111
                ora #'0'
                cmp #'9'+1
                bcc pr
                adc #'A'-'9'-2
pr              sta ($80),y
                dey
                rts
```

Rozdział 9

Procesor architektury x86

> Ja Państwu nie każę pisać w assemblerze.
>
> (4D 5A)

Najpopularniejszą 32-bitową architekturą procesorów spotykanych w komputerach jest architektura x86 (dokładniej x86-32, zwana także IA-32). Jest ona zaliczana do kategorii CISC (akronim od ang. *Complex Instruction Set Computing*), tzn. zawiera złożone instrukcje, wiele trybów adresowania, bezpośrednie odwołania do pamięci.

Historia tej architektury rozpoczyna się w roku 1985, kiedy wychodzi procesor 80386, rozszerzający 16-bitową architekturę x86-16 (8086 - 80286). Architektura była rozwijana i ulepszana, np. przez przetwarzanie potokowe, superskalarność, przewidywanie rozgałęzień[1], wielordzeniowość.

9.1 Rejestry

W podstawowej wersji architektury dostępne są następujące rejestry ogólnego przeznaczenia:

Rejestry danych

EAX - akumulator

EBX - rejestr bazowy

ECX - rejestr licznika

EDX - rejestr danych

Rejestry adresowe

ESI - rejestr źródła

[1] Jest to obecnie gorący temat, w związku z atakami Spectre i Meltdown, które wykorzystują lukę w tym ulepszeniu do uzyskania nieuprawnionego dostępu do danych, zobacz: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`

EDI - rejestr celu

EBP - wskaźnik podstawy stosu

ESP - wskaźnik szczytu stosu

Rejestry segmentowe

Zestaw 16-bitowych rejestrów służących do określania adresów segmentów w trybie rzeczywistym oraz jako selektory w trybie chronionym

CS - rejestr segmentu kodu

DS - rejestr segmentu danych

SS - rejestr segmentu stosu

ES, FS, GS - rejestry pomocnicze segmentu danych

Rejestry specjalnego przeznaczenia

EIP - licznik programu

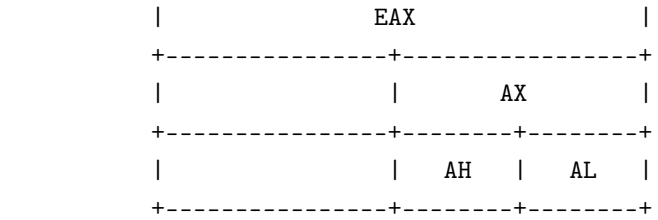
EFLAGS - rejestr stanu procesora

inne - rozszerzenia (FPU/MMX, SSE), rejestry kontrolne

Rejestry mniejszych rozmiarów

Z rejestrów danych, adresowych, EIP oraz EFLAGS można korzystać jak z rejestrów 16-bitowych (rejestry takie oznacza się przez pominięcie litery E, tzn. EAX $\mapsto$ AX.

Dodatkowo, do rejestrów danych i adresowych można się odwoływać do starszego i młodszego bajtu 16-bitowego rejestru, oznaczanych odpowiednio literami H i L, tzn. starszy bajt 16-bitowej części akumulatora, to AH.



Schemat: Subrejestry

9.2 Flagi procesora

Jednym z ważniejszych dla przebiegu wykonania kodu rejestrem jest rejestr EFLAGS. Przechowywany jest w nim stan procesora, głównie zmieniany przez instrukcje, porównania itp.

Tablica 9.1: Znaczenie bitów rejestru EFLAGS

bit	skrót	opis
0	CF	flaga przeniesienia
1	1	zarezerwowane
2	PF	flaga parzystości
3	0	zarezerwowane
4	AF	flaga wyrównania
5	0	zarezerwowane
6	ZF	flaga zera
7	SF	flaga znaku
8	TF	flaga pracy krokowej
9	IF	flaga przerwań
10	DF	flaga kierunku
11	OF	flaga przepełnienia
12,13	IOPL	poziom uprzywilejowania (tryb chroniony)
14	NT	flaga zadania zagnieżdżonego
15	0	zarezerwowane
16	RF	flaga wznowienia
17	VM	flaga trybu wirtualnego 8086
18	AC	sprawdzenie wyrównania
19	VIF	flaga przerwania wirtualnego
20	VIP	oczekujące przerwanie wirtualne
21	ID	identyfikacja
22 - 31	0	zarezerwowane

9.2.1 Ustawianie flag przez operacje

Carry

Ustawiana, gdy przy operacji wyjdziemy poza zakres.
Przykład:

```
mov eax, 0xFFFFFFFF
add eax, 1 ; CF = 1, EAX = 0
```

9.2.2 Overflow

Flaga overflow jest ustawiana przy dodawaniu i odejmowaniu równolegle do flagi carry, i mówi ona o "anomaliach" przy wykonywaniu operacji na liczbach ze znakiem. Jak flaga carry mówiła, że wynik nie zmieścił się w zakresie, tak flaga overflow mówi o tym, że doszło do anomalii znaku, tzn. patrzymy jedynie na bity znaku (najbardziej znaczące bity):

- $1 + 1 \mapsto 0$, czyli dodajemy dwie liczby ujemne i otrzymujemy dodatnią

- $0 + 0 \mapsto 1$, czyli dodajemy dwie dodatnie i otrzymujemy ujemną

Przykład:

```
mov al, 127
add al, 127 ; AL = 254 (-2 w U2), OF = 1
```

9.3 Tryby adresowania

Procesory architektury x86 umożliwiają kilka trybów adresowania instrukcji (por. kategoria CISC), :

- Rejestrowe - operandem jest zawartość rejestru
- Bezpośrednie - bezpośrednio podana wartość operandu
- Pośrednie - wartość operandu znajduje się w pamięci pod podanym adresem
- Pośrednie z przemieszczeniem - jak wyżej, z tym że można dodać stałą do adresu
- Pośrednie Rejestrowe - adres pod którym znajduje się wartość znajduje się w rejestrze (rejestry EBX, EBP, ESI, EDI)
- Bazowe z przemieszczeniem - korzystanie z adresu jako wskaźnika na pierwszy element tablicy (por. tablice w C)
- Bazowo-indeksowe - korzystając z dwóch rejestrów wyliczamy adres, pod którym jest wartość operandu
- Bazowo-indeksowe z przemieszczeniem - jak wyżej, plus stała

```
mov eax, ebx              ; Rejestrowe
mov eax, 1                ; Bezpośrednie
mov eax, [02E0h]          ; Pośrednie
mov eax, tablica[2]       ; Pośrednie z przemieszczeniem
mov eax, [edi]            ; Pośrednie rejestrowe
mov eax, tablica[ebx]     ; Bazowe z przemieszczeniem
mov eax, [ebx + edi]      ; Bazowo-indeksowe
mov eax, [ebx + edi + 10] ; Bazowo-indeksowe z przemieszczeniem
```

9.3.1 Etykiety

Dzięki możliwości stosowania etykiet w asemblerze, zyskujemy przede wszystkim więcej elastyczności i odporności na modyfikację kodu (dodanie jednej linijki na początku programu, przy braku etykiet wymagałby zmiany wszystkich adresów). Dodatkowo niektóre asemblery (np. NASM) umożliwiają tzw. etykiety lokalne, czyli w zakresie od poprzedniej etykiety nielokalnej. Dzięki temu popularne etykiety, jak `.loop`, `.else`, czy `.end` moga być wykorzystane wielokrotnie w jednym programie.

Przykład

```
funkcja:                ; etykieta
        mov eax, 0
        mov ecx, 5
    .loop:              ; etykieta lokalna
        cmp ecx, 0
        jz  .end        ; skok do etykiety lokalnej z zakresu (poza nim funkcja.end)
        add eax, ecx
        jmp .loop
    .end:
        ret
```

9.4 Wybrane rozkazy procesora

Przyjmujemy oznaczenia: `reg` - rejestr, `addr` - adres

9.4.1 Przesyłanie danych

`MOV dest, src` - kopiowanie danych z *src* do *dest*

`PUSH src` - wstawienie na szczyt stosu wartości *src*

`POP dest` - zdejmuje wartość ze stosu i wstawia do *dest*

`LEA reg, addr` - do *reg* ładuje adres *addr*

9.4.2 Arytmetyczne i logiczne

`ADD dest, src` - dodawanie (dest := dest + src)

`ADC dest, src` - dodawanie z przeniesieniem (dest := dest + src + CF)

`SUB dest, src` - odejmowanie (dest := dest - src)

`SBB dest, src` - dodawanie z przeniesieniem(dest := dest - src - CF)

`SHL dest, (const|cl)` - przesunięcie bitowe w lewo

`SHR dest, (const|cl)` - przesunięcie bitowe w prawo

`MUL src` - mnożenie bez znaku (EDX:EAX := EAX * src)

`IMUL src` - mnożenie ze znakiem (EDX:EAX := EAX * src)

`IMUL reg, src` - mnożenie ze znakiem (reg := reg * src)

`DIV src` - dzielenie ze znakiem (EAX := (int)(EDX:EAX / src); EDX := EDX:EAX mod src)

`IDIV src` - dzielenie ze znakiem (EAX := (int)(EDX:EAX / src); EDX := EDX:EAX mod src)

`AND dest, src` - bitowe 'i' (dest := dest & src)

`OR dest, src` - bitowe 'lub' (dest := dest | src)

`XOR dest, src` - alternatywa wykluczająca[2] (dest := dest ^ src)

`CMP dest, src` - porównanie

9.4.3 Skoki

`JMP addr` - skok bezwarunkowy

`JE addr` - skok, gdy równe

`JNE addr` - skok, gdy nierówne

`JZ addr` - skok, gdy ustawiony bit zera

`JNZ addr` - skok, gdy wyczyszczony bit zera

`JG addr` - skok, gdy większe

`JGE addr` - skok, gdy większe lub równe

`JL addr` - skok, gdy mniejsze

`JLE addr` - skok, gdy mniejsze lub równe

`LOOP addr` - `ECX := ECX - 1`, skok, gdy `ECX` $\neq$ `0`

`CALL addr` - wstaw na stos adres następnej instrukcji i skocz pod `addr`

`RET` - pobierz ze stosu adres powrotu i skocz

9.4.4 Inne

`IN port` - pobierz dane z portu I/O

`OUT port` - wyślij dane do portu I/O

`INT num` - wywołaj przerwanie o numerze `num`

9.5 Sekcje

Jak wiadomo, program nie ma jednolitej struktury. Oprócz kodu wykonywalnego, przechowywane są także informacje dodatkowe oraz dane wykorzystywane przez program.

9.5.1 .text

Sekcja ta zawiera kod programu, oczywiście, nie jest zabronione odczytywanie danych z tej sekcji, jednak najczęściej nie można zapisać danych do tej sekcji.

[2] często stosowane do zerowania rejestru: `xor eax, eax`

9.5.2 .data

Sekcja zawiera zainicjalizowane dane wraz z ich wartościami. Są one przechowywane w pliku obiektowym. Do tej sekcji można zapisywać i czytać.

9.5.3 .bss

Sekcja podobna do `.data`, jednak przechowująca dane niezainicjalizowane, tzn. przechowuje w pliku obiektowym informację o rozmiarze danych.

9.6 Stos

Jednym z ważniejszych regionów w pamięci jest stos. Stos jest miejscem w pamięci (zwykle początek w wysokich adresach, stos "rośnie" w dół), na którym możemy operować jedynie na elemencie ze szczytu (przy użyciu tej pamięci jako stosu, bez problemu można się odwoływać do tej pamięci w sposób "klasyczny".

Opis stosu jest tworzony przez parę rejestrów : `EBP` i `ESP`. Pierwszy wskazuje na podstawę stosu (jest to przydatne m. in. do mechanizmu zmiennych lokalnych), natomiast `ESP`, wskazuje na najwyższy element.

Ze stosu możemy korzystać z instrukcji `push` i `pop`, ktore wykonują (znacznie szybciej niż poniższy kod) operacje odpowiednio wstawienia i zdjęcia ze stosu elementu:

```
push_eax:              ; równoważne push eax
    sub esp, 4
    mov [esp], eax

pop_eax:               ; równoważne pop eax
    mov eax, [esp]
    add esp, 4
```

9.7 Konwencje wywołań podprogramów

Kiedy tworzymy podprogram (funkcję, procedurę) zwykle potrzebujemy przekazać jej jakieś parametry oraz odebrać wynik jej działania. Jak wiadomo, programiści nie lubią się męczyć, dlatego tworzą oni standardy, protokoły komunikacji, interfejsy API, które ujednolicają komunikację między modułami.

Gdyby nie istniał wspólny sposób komunikacji pomiędzy komponentami, tworzenie nowych porcji kodu byłoby bardzo trudne. Na poziomie kodu maszynowego, najważnejszym "protokołem" jest konwencja wywołania (*ang. calling convention*), która opisuje, w jaki sposób przekazywać parametry, gdzie znajduje się wynik oraz które rejestry są zachowywane podczas wywołania. Dodatkowo, konwencja wywołania określa, kto sprząta stos po zakończeniu procedury (niesie to bardzo poważne implikacje).

9.7.1 Ramka stosu

Podczas analizy kodu generowanego przez GCC w większości funkcji można natrafić na następujący ciąg instrukcji:

```
push    ebp
mov     ebp, esp
```

Powyższy kod ustawia tak zwaną ramkę stosu, tzn. odkłada na stos poprzedni adres podstawy stosu, a obecny szczyt uznajemy za podstawę stosu w kontekście naszej funkcji.

Następnie następuje przygotowanie miejsca dla zmiennych lokalnych:

```
sub     esp, X
```

W ten sposób zarezerwowaliśmy X bajtów. Do odwoływania się do zmiennych będziemy wykorzystywać rejestr `ebp` z odpowiednim przesunięciem.

Oczywiście, bezpośrednio przed powrotem z funkcji, musimy przywrócić poprzedni stan stosu, więc wywołujemy operacje przeciwne, w odwrotnej kolejności (pomijając rezerwację miejsca na zmienne, ponieważ szczyt stosu i tak zostaje odrzucony):

```
mov     esp, ebp
pop     ebp
```

9.7.2 Wybrane konwencje wywołań

Główny podział konwencji wywołań zależy od tego, kto czyści stos. W przypadku konwencji, w których wywołujący czyści stos, mamy możliwość realizacji funkcji o zmiennej liczbie argumentów (*ang. variadic function*). Oczywiście istnieje wiele konwencji, które są stosowane przez różne kompilatory, środowiska, a szczególne różnice widać pomiędzy systemami operacyjnymi.

Wywołujący czyści

CDECL

Główna konwencja wywołań, powiązana z językiem C (C DECLaration) oraz przyjęta jako główna konwencja dla systemów rodziny UNIX.

Argumenty od prawej do lewej

Rejestry niezachowane EAX, ECX, EDX

Rejestry zachowane pozostałe

Wynik EAX

Wywoływany czyści

STDCALL

Konwencja bardzo podobna do CDECL, stosowana głównie przez Windowsowe WINAPI. Główna różnica polega na tym, że to wywołana funkcja czyści stos

Argumenty od prawej do lewej

Rejestry niezachowane EAX, ECX, EDX

Rejestry zachowane pozostałe

Wynik EAX

9.7.3 Wywołanie funkcji

Aby zebrać wszystkie informacje przeanalizujmy prostą funkcję, stosującą konwencję wywołań **cdecl**, która będzie miała dwie zmienne lokalne typu int (32-bitowe) oraz przyjmie dwa argumenty. Funkcja ma zwrócić sumę argumentów. Opis adresów po ustawieniu ramki stosu:

`ebp+12` ↦ drugi argument

`ebp+8` ↦ pierwszy argument

`ebp+4` ↦ adres powrotu

`ebp+0` ↦ stara wartość ebp

`ebp-4` ↦ pierwsza zmienna

`ebp-8` ↦ druga zmienna

Listing funkcji:

```
funkcja:                    ; int f(int a, int b) {
    push    ebp
    mov     ebp, esp
    sub     esp, 8          ;     int x,y;

    mov     eax, [ebp+8]    ;     x = a;
    mov     [ebp-4], eax
    mov     eax, [ebp+12]   ;     y = b;
    mov     [ebp-8], eax

    mov     eax, [ebp-4]    ;     return x+y;
    add     eax, [ebp-8]

    mov     esp, ebp        ; }
    pop     ebp
    ret
```

Listing wywołania:

```
    ; ...
    push    4
    push    3
    call    funkcja         ; f(3, 4)
    add     esp, 8
    ; ...
```

9.8 Tryb rzeczywisty

Tryb rzeczywisty jest to 16-bitowy tryb, utrzymywany ze względu na kompatybilność wsteczną ze starszymi procesorami. Dlatego też, wszystkie procesory architektury x86 startują w trybie rzeczywistym i trzeba jawnie przejść w tryb chroniony. Obecnie, rzadko kiedy wykorzystuje się tryb rzeczywisty, ponieważ mamy dostęp jedynie do 20-bitowej przestrzeni adresowej (1 MB), nie ma wsparcia wielozadaniowości ani ochrony pamięci (swobodny dostęp do całej pamięci). Przez uruchomienie procesora w trybie rzeczywistym, BIOS oraz bootloadery pracują w trybie rzeczywistym, gdzie bootloadery starają się bardzo szybko przejść do trybu chronionego.

9.9 Tryb chroniony

Tryb chroniony jest bardziej rozbudowanym trybem. Jest on całkowicie 32-bitowy, pozwala zaadresować 4 gigabajty pamięci, a przede wszystkim udostępnia nam pamięć wirtualną i mechanizm stronicowania pamięci. Dodatkowo udostępnia on możliwość wirtualizacji procesora 8086 (tryb wirtualny 8086) oraz mechanizmy sprzętowej wielozadaniowości. Większość rozszerzeń zbioru instrukcji jest dostępna jedynie w trybie chronionym.

9.9.1 Przejście w tryb chroniony

Aby poprawnie przejść do trybu chronionego należy:

- Zablokować przerwania (instrukcja `cli`)
- Odblokować linię A20
- Stworzyć i załadować GDT (Global Descriptor Table) z deskryptorami segmentów kodu, danych i stosu
- Ustawić zerowy bit rejestru kontrolnego CR0

9.10 Przykładowe programy i zadania

Zadanie 1. Przeanalizuj krótki program w asemblerze x86.

```
MOV AL, L1
MOV BL, L2
CLC
ADC AL, BL
```

Uzupełnij w tabeli wartość flag po wykonaniu programu.

L1	L2	CF (carry flag)	OF (overflow flag)	ZF (zero flag)
-1	1			
150	-28			
127	-128			

Rozwiązanie. • `L1 = -1, L2 = 1` Jak wiemy, procesor nie zna bezpośrednio pojęcia 'znaku' liczby i operuje jedynie na wartościach bitowych. Zatem liczby przekształcamy zgodnie z kodowaniem U2 i uzyskujemy:

`AL = 0xFF, BL = 0x01`

Dodajemy je (jak dwie liczby bez znaku) i otrzymujemy:

`wynik = 0x100 => AL = 0x00, CF = 1, OF = 0, ZF = 1`

- `L1 = 150, L2 = -28` Wstawienie 150 interpretujemy jako wstawienie liczby bez znaku:

$$150 = (10010110)_2 = (96)_{16}$$

Drugą liczbę przekształcamy zgodnie z kodowaniem U2:

$$-28 = -(0001110)_2 = (1110010)_{U2} = (72)_{16}$$

Otrzymane wartości wstawiamy do rejestrów i obliczamy wynik

`AL = 0x96, BL = 0x72`

`wynik = 0x108 => AL = 0x08, CF = 1, OF = 1, ZF = 0`

- `L1 = 127, L2 = -128` Wstawienia dokonujemy analogicznie

`AL = 0x7F, BL = 0x80`

Obliczamy sumę

`wynik = 0xFF => AL = 0xFF, CF = 0, OF = 0, ZF = 0`

Zadanie 2. Przeanalizuj krótki program w asemblerze x86 i uzupełnij wynik. Liczby są zapisane w U2.

```
        XOR EAX, EAX
        NOT EAX
PETLA:  ADC EAX, 4
        JC PETLA
```

WYNIK: 0x_ _ _ _ _ _ _ _ . Dlaczego taki?

Rozwiązanie.
Rozpoczynamy nie znając wartości EAX, jednak już pierwsza linijka zeruje rejestr.

```
XOR EAX, EAX; EAX = 0
```

```
NOT EAX; EAX = 0xFFFFFFFF
```

Następnie wykonujemy dalej

```
ADC EAX, 4; EAX = 0x00000003; CF = 1
```

Flaga Carry jest ustawiona więc wykonujemy skok

```
JC PETLA
```

Flaga Carry jest ustawiona, więc następna linijka dodaje 4 + CF, czyli 5

```
ADC EAX, 4; EAX = 0x00000008; CF = 0
```

Flaga Carry jest zgaszona więc nie wykonujemy skoku i kończymy wykonanie programu.
Ostatecznie, odpowiedzią jest `0x00000008`

9.11 Koprocesor zmiennoprzecinkowy x87

9.11.1 Rejestry

W koprocesorze x87 mamy do dyspozycji osiem rejestrów 80 bitowych. Rejestry te, oznaczane przez `st0`, `st1`, itd., działają jak stos i jako programista mamy tylko możliwość przekazywania nowych wartości na stos oraz pobierania tych już tam będących za pomocą odpowiednich instrukcji oraz żądanie wykonania pewnych działaś arytmetycznych. Należy pamiętać, że z uwagi na to, iż rejestry są 80 bitowe, koprocesor, wewnętrznie, wykonuje wszystkie działania używając rozszerzonej precyzji.

9.11.2 Flagi koprocesora

Koprocesor posiada 16 bitowy rejestr statusu w którym znajdują się poniższe informacje:

- Bit 12 to tak zwany bit kontroli nieskończoności, który służył do zachowania kompatybilności z Intelem 287
- Bity 10 oraz 11 kontrolują metodę zaokrąglania. Tryby to kolejno (od 0 do 3): do najbliższej liczby całkowitej, w dół, w górę, obcięcie
- Bity 8 oraz 9 kontrolują precyzję obliczeń. Tryby to kolejno: 24, pominięty, 53 bity, 64 bity
- Pozostałe bity zawierają wiele informacji o przebiegu obliczeń. Idąc od zera mamy tam flagi: błędu operacji, denormalizacji, dzielenia przez zero, przepełnienia, niedomiaru oraz precyzji

9.11.3 Status koprocesora

Koprocesor posiada również 16 bitowy rejestr statusu, w którym znajdują się poniższe informacje (począwszy od bitu 0):

- Flaga niewłaściwej operacji
- Flaga denormalizacji
- Flaga Dzielenia przez zero
- Flaga nadmiaru
- Flaga niedomiaru
- Flaga precyzji
- Flaga błędu stosu
- Flaga podsumowania wyjątku
- Flagi C0, C1 oraz C2 (o nich później)
- Wskaźnik szczytu stosu (3 bity)
- Flaga C3
- Flaga informująca, że koprocesor jest zajęty

9.11.4 Flagi porównania

Flagi C0, C1, C2 oraz C3 wykorzystuje się głównie przy porównywaniu wartości. Na podstawie ciągu C3C2C1C0 określamy warunek w następujący sposób:

Tablica 9.2: Słownik flag porównania x87

C3C2C1C0	**Warunek**
00X0	ST > źródło
00X1	ST < źródło
10X0	ST = źródło
11X1	ST lub źródło niezdefiniowane

9.11.5 Oznaczenia typów danych

- [mem32] - pojedyncza precyzja
- [mem64] - podwójna precyzja
- [mem80] - rozszerzona precyzja
- [mem] - dowolna z powyższych

9.11.6 Wybrane rozkazy koprocesora

Instrukcje kontrolne to:

- FWAIT - Program czeka, aż koprocesor zakończy obliczenia
- FNINIT - Inicjalizuje koprocesor. Wymagane przed użyciem
- FINIT - Równoważnie z wywołaniem FNINIT oraz FWAIT

Instrukcje manipulujące stosem to:

- FLD [mem] - Ładuje liczbę zmiennoprzecinkową na stos
- FILD [mem] - Ładuje liczbę całkowitą na stos
- FST [mem32/64/80] - zapisuje do pamięci wartość w st0
- FSTP [mem32/64/80] - zapisuje do pamięci wartość w st0 oraz zdejmuje ją ze stosu
- FIST [mem16/32] - zapisuje do pamięci wartość w st0 obciętą do liczby całkowitej
- FIST [mem16/32] - zapisuje do pamięci wartość w st0 obciętą do liczby całkowitej oraz zdejmuje ją ze stosu
- FXCH st(i) - zamienia st0 z st(i) gdzie $i \in \{1, \ldots, 7\}$

- FLDZ - ładuje zero na stos
- FLD1 - ładuje jedynkę na stos
- FLDPI - ładuje liczbę pi na stos
- FLDL2T - ładuje $\log_2(10)$ na stos
- FLDL2E - ładuje $\log_2(e)$ na stos
- FLDLG2 - ładuje $\log_{10}(2)$ na stos
- FLDLN2 - ładuje $\ln(2)$ na stos

Instrukcje dodawania to:

- FADD [mem32/64] - Wykonuje działanie $st0 = st0 + [mem]$
- FADD st(i) st0 - Wykonuje działanie $st(i) = st(i) + st0$
- FADD st0 st(i) - Wykonuje działanie $st0 = st0 + st(i)$
- FADD /bez argumentów/ - Wykonuje działanie $st1 = st1 + st0$
- FADDP st(i) st0 - Wykonuje działanie $st(i) = st(i) + st0$ oraz zdejmuje ze stosu
- FADDP /bez argumentów/ - Wykonuje działanie $st1 = st1 + st0$ oraz zdejmuje ze stosu

Instrukcje odejmowania FSUB oraz FSUBP działają analogicznie do dodawania. Z uwagi na to, że w odejmowaniu kolejność ma znaczenie mamy również do dyspozycji instrukcje FSUBR oraz FSUBRP działające następująco:

- FSUBR [mem32/64] - Wykonuje działanie $st0 = [mem] - st0$
- FSUBR st(i) st0 - Wykonuje działanie $st(i) = st0 - st(i)$
- FSUBR st0 st(i) - Wykonuje działanie $st0 = st(i) - st0$
- FSUBR /bez argumentów/ - Wykonuje działanie $st1 = st0 - st1$
- FSUBRP st0 st(i) - Wykonuje działanie $st0 = st(i) - st0$ oraz zdejmuje ze stosu
- FSUBRP /bez argumentów/ - Wykonuje działanie $st0 = st1 - st0$ oraz zdejmuje ze stosu

Mamy również do dyspozycji kilka dodatkowych działań:

- FMUL - Mnożenie. Składnia jak przy dodawaniu
- FDIV - Dzielenie. Składnia jak przy dodawaniu
- FDIVR - Dzielenie. Składnia jak przy odejmowaniu odwrotnym

- FPREM - Wykonuje działanie $st0 = st0 mod st1$ (Nowsza wersja tej instrukcji to FPREM1)
- FABS - Wykonuje działanie $st0 = |st0|$
- FCHS - Wykonuje działanie $st0 = -st0$
- FSQRT - Wykonuje działanie $st0 = \sqrt{st0}$
- FRNDINT - Wykonuje zaokrąglenie $st0 = (int)st0$
- FSCALE - Wykonuje działanie $st0 = st0 \cdot 2^{st1}$. Jeżeli st1 nie jest liczbą całkowitą to jest do niej automatycznie obcinany przed wykonaniem tej instrukcji

Rozdział 10

Procesor ARM

> Battle Royale to gatunek gier, który zapoczątkowany został przez mod do gry ARMA 2... O przepraszam, to nie to okno.
>
> (PaKman)

ARM (Advanced RISC Maschine) jest architekturą procesorów z kategorii RISC - zmniejszony zbiór instrukcji, zwiększona ilość rejestrów. ARM jest obecnie architekturą bardzo popularną w systemach wbudowanych, a także w urządzeniach mobilnych.

10.1 Tryby pracy

User mode (usr) - normalny tryb wykonywania kodu

FIQ mode (fiq) - obsługa szybkich przerwań (np. obsługa transferu danych)

IRQ mode (irq) - obsługa przerwań

Supervisor mode (svc) - tryb chroniony dla systemu operacyjneg

Abort mode (abt) - tryb w który przechodzi procesor po wykonaniu instrukcji abort

Undefined mode (und) - tryb wywołany przez wykonanie niezdefiniowanej instrukcji

10.2 Rejestry

W przeciwieństwie do architektury x86, ARM posiada więcej rejestrów:

- Rejestry ogólne: **R0 - R7**
- Rejestry ogólne zachowywane podczas szybkich przerwań: **R8 - R12**
- Rejestry stosu i powrotu: R13 (SP - Stack Pointer), R14 (LR - Link Register)
- Licznik programu: R15 (PC - Program Counter)

- Rejestr stanu: CPSR (Current Processor State Register)

Część trybów pracy posiada własne "kopie" rejestrów:

FIQ - rejestry R8 - R14

SVC, ABT, IRQ, UND - rejestry R13, R14

10.3 Flagi procesora

Obecne flagi są ustawione w rejestrze stanu procesora: CPSR.

Tablica 10.1: Znaczenie bitów rejestru CPSR

bit	**skrót**	**opis**
0-4	M	tryb procesora
5	T	Thumb state - alternatywny zestaw instrukcji (kodowany na 16 bitach)
6	F	zablokowanie szybkich przerwań
7	I	zablokowanie przerwań
8	A	"imprecise data abort disable bit"[1]
9	E	bit kolejności bajtów
10 - 15 i 25-26	IT	bity stanu if-then
16-19	GE	bity "większy lub równy"
24	J	zestaw instrukcji Jazelle
27	Q	bit niedomiaru / saturacji
28	V	bit oVerflow
29	C	bit Carry
30	Z	bit zera
31	N	bit wartości ujemnej

10.3.1 Ustawianie flag

Carry

dodawanie(ADD, CMN) Flaga carry zostanie zapalona, jeśli liczba nie zmieści się w zakresie

odejmowanie(SUB, CMP) Flaga carry zostanie zapalona, jesli **NIE** było ‘'pożyczenia bitu" w odejmowaniu. Jeśli bit został "pożyczony" flaga jest zgaszona

przesunięcia Flaga będzie ustawiona na ostatni z bitów które przesunęliśmy

Overflow

Flaga overflow zostanie zapalona, gdy wynik dodawania lub odejmowania jest $\geqslant 2^{31}$ lub $< -2^{31}$

10.4 Składnia instrukcji

Każda instrukcja w procesorze ARM (w podstawowym zestawie instrukcji) jest kodowana na 32 bitach. Dodatkowo pierwsze 4 bity są przeznaczone na kodowanie warunku, tzn. każda instrukcja jest instrukcją warunkową, co umożliwia znaczne uproszczenie programów i zmniejszenie ilości rozgałęzień, np.

```
if (r1 > r2)
        r1 -= r2
else
        r2 -= r1
```

Można przetłumaczyć na

```
cmp    r1, r2      ; porównaj r1 r2
subgt  r1, r1, r2  ; jeśli r1 > r2, r1 -= r2
suble  r2, r2, r1  ; jeśli r1 <= r2, r2 -= r1
```

10.5 Wybrane rozkazy procesora

Pamiętajmy, że każda instrukcja może zawierać warunek wykonania

`LDR reg, addr` - załaduj adres *addr* do rejestru *reg*

`LDR reg1, [reg2]` - załaduj wartość spod adresu zawartego w *reg2* do rejestru *reg1*

`STR reg1, [reg2]` - zapisz zawartość rejestru reg1 pod komórkę pamięci o adresie zawartym w rejestrze reg2

`MOV reg1, reg2` - skopiuj zawartość reg2 do reg1

`B addr` - skok pod adres addr

`BL addr` - skok pod adres addr, skopiuj zawartość r15 do r14

`CMP reg, (reg| const)` - porównaj wartości i ustaw flagi

`ADD dest, src, operand` - dodaj operand do src i przechowaj w dest

`SUB dest, src, operand` - odejmij operand od src i przechowaj w dest

Hint: Instrukcje ADD i SUB mają odmiany ustawiające flagi, odpowiednio: ADDS, SUBS.

Więcej informacji na temat architektury ARM można znaleźć m. in. na OSDev Wiki, `https://wiki.osdev.org/ARM_Overview`

10.6 Przykładowe programy i zadania

Zadanie 1 (3pt)**.** Procesor o architekturze ARM wykonuje następujący kod:

```
LDR    R1, =STALA_A
LDR    R2, =STALA_B
SUBS   R0, R1, R2
```

Jak po wykonaniu rozkazu SUBS ustawione zostaną bity przeniesienia C (ang. carry), nadmiaru V (ang. overflow) i zera Z (ang. zero) dla poniższych wartości STALA_A, STALA_B:

STALA_A	STALA_B	przeniesienie C	nadmiar V	zero Z
-1	1			
150	-28			
127 (?)	-128 (?)			

Zadanie 2 (3pt)**.** Jaka liczba (używamy reprezentacji U2) znajduje się w rejestrze R0 po wykonaniu poniższego kodu?

```
        MOV r0, #0
        MOV R1, #10
again   ADD R0, R0, R1
        SUBS R1, R1, #1
        BNE again
```

Wartość rejestru R0 = 0x _ _ _ _ _ _ _ _ (szesnastkowo) po wykonaniu kodu. Opisz krótko dlaczego.

Rozdział 11

Technologia CUDA

CUDA to po prostu opowieści, w które wierzymy, gdyż pragniemy, żeby były prawdziwe.

(Dan Brown)

Karty graficzne są urządzeniami z wysoce wyspecjalizowaną architekturą, ukierunkowaną na wykonywanie równolegle ogromnej ilości wielu prostych, niezależnych od siebie obliczeń. Przetwarzanie i obróbka grafiki i filmów, a także grafika w grach komputerowych jest szczególnym przykładem takich zadań. Dlatego też powstawały coraz mocniejsze procesory graficzne, które pozwalały na uzyskiwanie coraz lepszych efektów.

Wiele z problemów algorytmicznych niezwiązanych z grafiką także pozwala na podzielenie zadań na w miarę niezależne od siebie części, przez co możliwe jest wykonywanie ich na wielu rdzeniach. Właśnie w tym celu, do przyspieszenia obliczeń inżynierskich, biologicznych czy astrofizycznych, rozwinęła się idea GPGPU - *General Purpose Computations on Graphical Processing Unit*. Jedną z ważniejszych technologii wspierających tą ideę jest wprowadzona w 2006 r. Nvidia CUDA.

11.1 Opis

Nvidia CUDA jest to API pozwalające na wykorzystanie GPU do obliczeń równoległych, niekoniecznie związanych z przetwarzaniem grafiki. Programy wykorzystujące CUDA są zwykle tworzone w rozszerzeniu/modyfikacji języka C++, 'CUDA C++', kompilowanym przy użyciu `nvcc`. Istnieje także możliwość wykorzystania CUDA w innych językach, przez użycie specjalnych bibliotek.

11.2 Przykładowy program

Omówmy przykładowy kod wykorzystujący CUDA, z poradnika od Nvidii [1].[2]

```
1 #include <iostream>
2 #include <math.h>
```

[1] Źródło: `https://devblogs.nvidia.com/even-easier-introduction-cuda/`

[2] Pomimo kilku godzin walki, nie udało mi się skonfigurować narzędzi CUDA w systemie Windows, opis oparty jest na podręczniku Nvidii - przypis autora

```
3
4  __global__
5  void add(int n, float* x, float* y) { // 1
6      for (int i = 0; i < n; i++) {
7          y[i] = x[i] + y[i];
8      }
9  }
10
11 int main(void) {
12     int N = 1 << 20;
13     float *x, *y;
14
15     cudaMallocManaged(&x, N*sizeof(float)); // 2
16     cudaMallocManaged(&y, N*sizeof(float));
17
18     // 3
19     for (int i = 0; i < N; i++) {
20         x[i] = 1.0f;
21         y[i] = 2.0f;
22     }
23
24     add<<<1, 1>>>(N, x, y); // 4
25
26
27     cudaDeviceSynchronize(); //5
28
29     float maxError = 0.0f;
30     for (int i = 0; i < N; i++) {
31         maxError = fmax(maxError, fabs(y[i] - 3.0f));
32     }
33
34     std::cout << "Max error: " << maxError << std::endl;
35
36
37     cudaFree(x); // 6
38     cudaFree(y);
39
40     return 0;
41 }
```

Opis działania:

1 flaga `__global__` oznacza funkcję CUDA, przeznaczoną do wywołania na GPU

2 rezerwujemy pamięć

3 wypełniamy tablice danymi

4 wywołujemy funkcję CUDA, (liczby w potrójnych nawiasach ostrokątnych oznaczają odpowiednio: ilość bloków i ilość wątków na blok)

5 musimy poczekać na wykonanie obliczeń

6 zwalniamy pamięć

Część II

Systemy operacyjne (Linux)

Rozdział 12

Programowanie wielowątkowe

– Puk puk.
– Wyścigi wątków!
– Kto tam?

(Znany żart o programowaniu wielowątkowym)

Domeną nowoczesnych systemów operacyjnych była wielozadaniowość systemu. Pozwala ona na wykonywanie różnych dla komputera bardziej, lub mniej, powiązanych ze sobą zadań. Poważnym komercyjnym przeskokiem było wydanie Windowsa 95, którym był pierwszym systemem pozwalający na uruchamianie kilku zadań jednocześnie. Ten rozdział będzie poświęcony właśnie wielozadaniowości w systemach uniksopodobnych zgodnych ze standardem POSIX.

12.1 Proces

Najbardziej atomowym bytem w systemie operacyjnym, jeśli chodzi o zadanie, jest proces. Procesem nazywamy instancję wykonywanego programu. Proces jest zarządzany przez planistę (ang. *job scheduler*), czyli wyspecjalizowany moduł jądra, który przydziela czas procesora na wykonanie kodu.

Proces może zostać utworzony głównie w dwóch scenariuszach: poprzez uruchomienie pliku wykonywalnego, bądź utworzenie go jako proces potomny do innego procesu.

12.1.1 Proces potomny

Proces potomny to proces, który jest uzależniony od procesu rodzica. Proces potomny często tworzy się w celach m. in.

- wykonania innego programu; często taki proces potomny ma niszczone środowisko wykonywania i podmieniane jest na środowisko wykonywanego programu,
- zwielokrotnienia działania jednego programu; widać to na przykładzie przeglądarki w oparciu o silnik Chromium.

12.1.2 Tworzenie procesów potomnych

By utworzyć proces potomny wykorzystuje się funkcję `int fork()`, która znajduje się po dołączeniu nagłówka dyrektywą `#include <unistd.h>`. Tworzy proces potomny, który wykonuje się od tej samej linii co wywołanie tej funkcji. By móc zidentyfikować, który proces jest procesem potomnym, funkcja `int fork()` zwraca liczbę całkowitą. Jeśli zostało zwrócone zero, wykonywany jest proces potomny, w przeciwnym wypadku jest to proces rodzicielski, bądź -1, gdy `int fork()` natrafił na błąd.

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int pid = fork();
    if (pid < 0) {
       printf("Coś się popsuło i nie było mnie słychać...\n");
       return -1;
    } else if (pid == 0) {
       printf("Proces potomny.\n");
    } else {
       printf("Proces-rodzic.\n");
    }
    return 0;
}
```

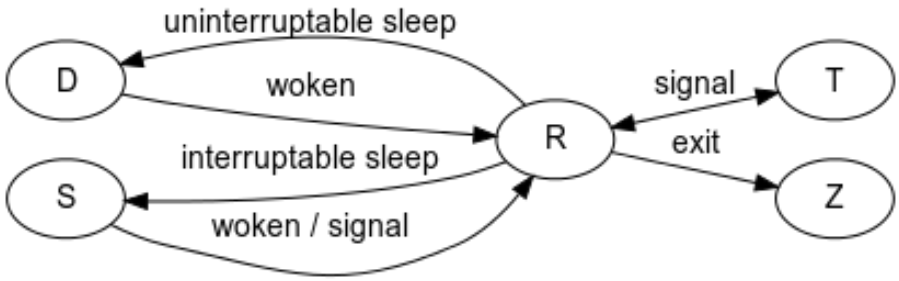
Listing 1: Przykład programu wykorzystującego funkcję `int fork()`.

12.2 Stany procesów

Każdy proces ma swój stan. By dowiedzieć się o stanie procesu z perspektywy powłoki tekstowej, można użyć polecenia `ps`. Tabela stanów jest przedstawiona jako tabela **??**.

12.2.1 Wysyłanie sygnału do procesu

By móc zmienić zachowanie procesu, wprowadzony został mechanizm sygnałów. Sygnał jest to odpowiednik przerwania tylko między procesem a jądrem. Sygnały wysyła się za pomocą funkcji `int kill(pid_t pid, int signum)` w pliku `#include <signal.h>`. By móc obsłużyć sygnał, należy zarejestrować tzw. *signal hanlder* za pomocą funkcji `signalhandler_t signal(int signum, signalhandler_t handler)`. Typ `signalhandler_t` jest niczym innym niż wskaźnikiem do funkcji; jest definiowany jako `typedef void (*sighandler_t)(int)`. Są sygnały, które nie da się „zagłuszyć”, m.in. jest to sygnał `SIGKILL`.



Rysunek 12.1: Diagram stanów dla procesu

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int signum) {
    if (signum == 2) {
        printf("Otrzymano SIGINT\n");
        exit(1);
    }
}

int main(int argc, char *argv[]) {
    int pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) {
        signal(2, sigint_handler);
        printf("Czekamy na sygnał od rodzica...\n");
        while (1);
    } else {
        sleep(1);
        if (kill(pid, 2) < 0) {
            perror("kill");
            return 1;
        }
    }
}
```

Listing 2: Przykładowe użycie funkcji `kill` oraz `signal`

Tablica 12.1: Tabela stanów procesu

Symbol stanu	Nazwa stanu	Komentarz
`R`	running	Oznacza działający proces, który wykonuje jakieś zadanie.
`D`	uninterruptible sleep	Oznacza stan spoczynku procesu, jednakże nie da się go przerwać (przykładowo podczas dostępu do wejścia/wyjścia).
`S`	interruptible sleep	Oznacza stan spoczynku procesu, który można przerwać (przykładowo podczas oczekiwania na ukończenie pewnego, innego zdarzenia).
`Z`	defunct/zombie	Jest to proces-zombie. Proces-zombie to taki proces potomny, który ukończył swoje działanie i proces-rodzic nie zaczekał na niego.
`T`	stopped	Jest to proces zatrzymany. Proces zatrzymany to taki proces, którego wykonanie zostało zatrzymane i może zostać wznowione przez sygnały.

12.2.2 Oczekiwanie na proces

Na procesy również możemy oczekiwać. Jest to dość powszechnie używana funkcjonalność — przykładowo inny proces może pobierać coś, kiedy my czekamy na cały pobrany plik. Ku temu wykorzystuje się funkcję `int waitpid(pid_t pid, int *status, int options)`, która przyjmuje ID procesu, na którego oczekujemy, wskaźnik na zmienną, która będzie zawierała ostatni status procesu, który spowodował zwrócenie funkcji, oraz opcje, które określają, dla jakich stanów funkcja ma skończyć działanie. Jej działanie polega na blokowaniu głównego procesu, dopóki nie zostanie zmieniony stan procesu. Po więcej informacji można odwiedzić `$ man 2 waitpid`.

12.3 Wątki

Wątki są lekkim odpowiednikiem procesu. Proces może składać się z kilku wątków. Do zarządzania wątkami będziemy wykorzystywać bibliotekę `pthread`. Więcej informacji można znaleźć w tym wyczerpującym i dość zrozumiałym poradniku `https://computing.llnl.gov/tutorials/pthreads/`.

12.3.1 Pamięć dzielona

Pamięć dzielona (współdzielona, ang. *shared memory*) jest pamięcią dostępną dla każdego wątku. Są to zazwyczaj zmienne globalne. Pamięć dzielona pozwala na komunikację między wątkami i wymianę danych.

Zazwyczaj, gdy mało doświadczony programista pragnie zabrać się za programowanie wielowątkowe, zapomina się o podstawowej zasadzie podczas programowania wielowątkowego, której niezastosowanie jej prowadzi do takich problemów jak...

12.3.2 Wyścigi wątków

Wyścigi wątków to anomalia polegająca na braku synchronizacji wątków w dostępie do wspólnego dobra. Przykładem może być zwiększanie zmiennej całkowitej o dwa, z użyciem dwóch wątków, które zwiększają tę zmienną o jeden. Docelowy scenariusz naszego modelu wygląda następująco:

- Wątek 1 ładuje tymczasowo zmienną do pamięci podręcznej
- Wątek 1 dodaje jeden do tymczasowej zmiennej
- Wątek 1 przypisuje tymczasową zmienną do zmiennej powiększanej
- Wątek 2 ładuje tymczasowo zmienną do pamięci podręcznej
- Wątek 2 dodaje jeden do tymczasowej zmiennej
- Wątek 2 przypisuje tymczasową zmienną do zmiennej powiększanej

Jednakże bez należytej staranności, również może wystąpić następujący scenariusz:

- Wątek 1 ładuje tymczasowo zmienną do pamięci podręcznej
- Wątek 2 ładuje tymczasowo zmienną do pamięci podręcznej
- Wątek 1 dodaje jeden do tymczasowej zmiennej
- Wątek 2 dodaje jeden do tymczasowej zmiennej
- Wątek 1 przypisuje tymczasową zmienną do zmiennej powiększanej
- Wątek 2 przypisuje tymczasową zmienną do zmiennej powiększanej

Wtedy tak naprawdę rezultatem będzie zwiększenie o jeden, co oczywiście nie jest poprawnym działaniem tego algorytmu. Dlatego stosuje się...

12.3.3 Synchronizacja wątków

Są różne mechanizmy synchronizacji wątków. Głównie wykorzystywane to:

- Semafor — stanowi mechanizm blokowania procesów i wątków. Semafor, tak jak w prawdziwym życiu, można opuszczać i podnosić. Opuszczenie semafora wiąże się z blokowaniem procesu, a podnoszenie ze zwolnienia blokady. Jest to blokada jednostronna, tzn. doświadcza ją tylko proces, który stosuje się do semafora. Na ogół rzadko wykorzystywane ze względu na ich złożoność.
- Wzajemne wykluczenie (ang. *mutex*) — prosty mechanizm synchronizacji polegający na tym, że kiedy blokada została ustanowiona, to inne wątki, które chcą przejść do sekcji krytycznej, czekają, aż zostaje zwolniona. Jest to blokada dwustronna.
- Zmienne atomowe — zmienne, których odczyt i zmiana odbywa się w sposób atomowy, to znaczy, niepodzielny.

Rozdział 13

Obsługa wejścia-wyjścia

> UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.
>
> (Dennis Ritchie, współtwórca systemu UNIX)

System wejścia-wyjścia w systemach uniksopodobnych został zaprojektowany w sposób najprostszy, to znaczy, w taki sposób by zapewnić pewien polimorfizm między bytami, które można traktować jak plik. Najmniejszym niepodzielnym bytem w takim systemie jest sam plik. Zanim zaczniemy rozważać, przedyskutujemy pliki.

13.1 Pliki

Abstrahując od szczegółów implementacyjnych w danym systemie plików, plikiem nazywamy dowolny byt w systemie plików. Za ten byt możemy rozumieć między innymi:

- zwyczajny plik binarny, bądź tekstowy,
- folder — w systemach uniksowych jest zrealizowany jako plik, który ma specjalną funkcjonalność — zawiera inne pliki,
- urządzenie blokowe — urządzenie, które przechowuje dane w sposób ciągły, może być to fragment pamięci RAM (np. pamięć ramki bufora graficznego, tzw. *framebuffer*), bądź dysk twardy, stacja dysków wymiennych, etc.
- ram dysk — specjalny rodzaj pamięci tymczasowej przechowywanej w pamięci RAM, szeroko stosowana w komputerach takich jak Amiga, czy Atari,
- inny zamontowany system plików — w Linuksie korzystamy z biblioteki FUSE, która pozwala nam na korzystanie z innego systemu plików z wewnątrz naszego obszaru roboczego (por. montowanie systemu FAT z pliku).
- nazwany potok (o którym wspomnę pod koniec rozdziału) — jest to plik, który działa jako kolejka FIFO

- gniazdko — plik będący strumieniem danych przesyłanych przez sieć komputerową.

Jak widać, plik jest dość ogólnym, polimorficznym bytem.

13.2 Hierarchia katalogów w /

W każdym systemie uniksopodobnym występuje przeważnie podobna hierarchia plików, która zostanie tutaj wymieniona. Cały system plików znajduje się w katalogu głównym, oznaczanym jako `/`. W katalogu głównym znajdują się takie foldery jak:

Tablica 13.1: Podział katalogu głównego w systemie Linux

`/bin`	Zawiera najbardziej potrzebne binarki, z których korzysta system operacyjny wraz z użytkownikiem (np. `ls`, `cd`, `rm`, itd.).
`/boot`	Zawiera pliki wymagane do uruchomienia systemu (np. dane z partycji EFI, bootloader, pliki konfiguracyjne bootloadera, spakowane obrazy jądra, *initramfs*, etc.).
`/dev`	Zawiera urządzenia (jako pliki) podłączone do komputera, są to urządzenia blokowe, ram dyski, stacje dysków, karty dźwiękowe, etc.
`/etc`	Zawiera pliki ustawień systemowych.
`/home`	Zawiera katalogi domowe użytkowników systemu,
`/lib`	Folder zawierający systemowe biblioteki dzielone.
`/media`	Zawiera foldery będące zamontowanymi systemami plików na urządzeniach wymiennych np. pendrive, albo płyta CD
`/mnt`	Równoważny do folderu `/media`, zależne od dystrybucji.
`/proc`	Wirtualny katalog zawierający informacje o uruchomionych procesach (zużycie danych, zużycie czasu procesora, por. aplikację `top`).
`/sbin`	Zawiera pliki wykonywalne poleceń, które mogą zostać wykonane tylko przez administratora systemu (`root`).
`/tmp`	Zawiera pliki tymczasowe, często ten folder jest ram dyskiem.
`/usr`	Zawiera dodatkowe programy zainstalowane przez zarządcę pakietów.
`/var`	Zawiera pliki, których treść ulega częstym zmianom, przykładowo może zawierać logi systemowe, czy skrypty CGI.

13.3 Urządzenia blokowe

Urządzenia blokowe są specyficznym typem pliku w systemie plików. Ich ogólna charakteryzacja zostanie zwięźle opisana w tym rozdziale.

Tablica 13.2: Przegląd ważnych urządzeń blokowych w Linuksie

`/dev/sdX`	Dyski twarde, gdzie `X` to poszczególna litera przydzielona przez podsystem jądra odpowiadający za obsługę urządzeń blokowych
`/dev/sdXn`	Woluminy dysku twardego (partycje), gdzie `X` to poszczególna litera oraz `n` jako cyfra przydzielona przez podsystem jądra odpowiadający za obsługę urządzeń blokowych
`/dev/dsp` lub `/dev/sound`	Plik będący wejściem dla podsystemu audio. Tylko dla starszych Linuksów bez podsystemu ALSA
`/dev/input`	Folder zawierający peryferyjne urządzenia wejścia takie jak klawiatury, myszki, PC speaker
`/dev/fdn`	Stacje dyskietek, gdzie `n` to liczba
`/dev/fbn`	Ramki bufora graficznego, gdzie `n` to liczba
`/dev/null`	Skrzynka pocztowa doktora P. S.
`/dev/random`	Urządzenie losowe wytwarzające entropię w sposób niedeterministyczny, ważne urządzenie dla generowania kluczy w szyfrowaniu asymetrycznym (np. RSA, ElGammal)
`/dev/urandom`	Urządzenie losowe, które w wypadku, kiedy nie będzie entropii do wykorzystania, zacznie wykorzystywać algorytmy pseudolosowe, niezalecane do wykorzystywania w zastosowaniach kryptograficznych
`/dev/cdrom`	Stacja dysków optycznych (CD, DVD, Blu-ray)

13.4 Zarządzanie plikami

Unix, przez swoją prostotę i też epokę wykształcił jeden z najbardziej wygodnych systemów zarządzania plikami, poprzez filozofię „rób dużo małych i wyspecjalizowanych programów", określając to w kontekście m. in. zarządzania plikami. Zostanie przedstawiony przegląd wybranych funkcjonalności pozwalających na zarządzanie plikami z powłoki oraz za pomocą wywołań systemowych w tabeli 13.3. By pozyskać więcej informacji, można odwiedzić podręcznik `man`.

13.5 Prawa dostępu

Każdy plik w systemie uniksopodobnym ma informację na temat dostępu do pliku. Linux przewiduje cztery klasy dostępu do pliku: *user*, *group*, *others* i *all*. Każda z nich charakteryzuje się tym, że:

user Dostęp do pliku ma użytkownik tego pliku.

group Dostęp do pliku mają użytkownicy należący do danej grupy użytkowników (o której wspomnę później).

others Dostęp do pliku mają użytkownicy, którzy nie należą do danej grupy użytkowników, ani nie są właścicielem tego pliku.

all Połączenie trzech powyższych.

Tablica 13.3: Przegląd programów pozwalających na zarządzanie plikami w powłoce tekstowej

`cd` *<katalog>*	Zmienia bieżący katalog roboczy w powłoce tekstowej. Również powraca do poprzedniego wybranego katalogu jeśli zostanie podane `--`.	`$ pwd`
		`/home/foo`
		`$ cd dir`
		`$ pwd`
		`/home/foo/dir`
		`$ cd --`
		`/home/foo`

`ls` [-alh] [*<katalog>*]	Wyświetla pliki w bieżącym katalogu.	`$ ls` `foo bar baz biz buzz`

`rm` [-r] *<plik lub katalog>*	Usuwa pliki, bądź katalogi. Wraz z opcją -r usuwa pliki w wybranym katalogu wraz z katalogiem.	`$ ls`
		`foo.img bar.img`
		`$ rm foo.img`
		`$ ls`
		`bar.img`

`touch` *<plik>*	Tworzy pusty plik, bądź zmienia datę modyfikacji istniejącego pliku na dzisiejszą.	`$ ls`
		`foo.img bar.img`
		`$ touch biz.buzz`
		`$ ls`
		`foo.img bar.img biz.buzz`

`chmod` *<tryb>* *<plik>*	Zmienia flagi dostępu dla danego pliku. Po więcej informacji należy zajrzeć do podręcznika użytkownika.	`$ chmod +x myscript.sh`
		`$ ./myscript.sh`
		`It works!`

`chown`	Zmienia właściciela pliku (grupę i użytkownika). Więcej informacji w podręczniku użytkownika.	`$ chown owner2:group2 fancy.rmm`
`cp`	Kopiuje plik.	`$ cp wdip.txt pasty`
`mv`	Przenosi plik. Również zmienia nazwę pliku.	`$ mv rmm.txt rafal.txt`
`dd`	Kopiuje surowo treść jednego pliku do drugiego. Więcej informacji w podręczniku.	`$ dd if=/dev/zero of=/dev/sda1 bs=512`
`(u)mount`	Odmontowuje/montuje system plików zamontowany w środowisku użytkownika.	`$ sudo mount /dev/sda1 /mnt/gentoo` `$ sudo umount /mnt/gentoo`
`mkdir`	Tworzy katalog. Z opcją `-p` tworzy ścieżkę katalogów.	`$ mkdir rzs` `$ mkdir -p /mnt/gentoo/boot/grub`

Również są cztery uprawnienia, które można nadać każdej z klas:

- `r` Uprawnienie do czytania pliku.
- `w` Uprawnienie do zapisywania pliku.
- `x` Uprawnienie do wykonywania pliku.
- `s` Tak zwana flaga *setuid*. Uprawnienie do eskalacji uprawnień danego pliku wykonywalnego do uprawnień właściciela pliku i grupy przypisanej do tego pliku. Specjalny przypadek flagi uprawnień do wykonania pliku.
- `t` Tak zwany *sticky bit*. Dany plik, bądź katalog, może być usuwany, bądź przemianowany tylko przez właściciela pliku, lub owego katalogu, lub procesu, który ma prawo do modyfikacji tego pliku, czy katalogu.

Użytkownik określający dostęp może dla każdej klasy określić odrębny dostęp (oprócz *all* z wiadomych przyczyn), przykładowo dla każdego może ustalić uprawnienie do czytania pliku za pomocą poleceń `chmod` i `chown`. Przykładowo

```
$ chmod a=r,g=rw rmm
```

ustawia dla wszystkich uprawnienia do czytania oraz tylko dla grupy ustawia uprawnienia do czytania i zapisywania pliku `rmm`. Natomiast

```
$ chmod o-w rmm
```

zabiera uprawnienia do zapisywania pliku przez osoby nienależące do przypisanej grupy w pliku, bądź nie są właścicielem tego pliku. Można również wykonać

```
$ chmod u+x lemie.sh
```

by nadać sobie uprawnienia do wykonania pliku, często pomija się literę `u`.

By móc sprawdzić, jakie uprawnienia są dla danego pliku, bądź plików, można użyć ku temu polecenia `ls -l` (flaga `-h` w przykładzie ma na celu tylko wypisać wielkości plików w wielokrotnościach SI):

```
$ ls -lh
drwxrw----   2 konrad  rzs 10M Apr 8  12:53 projekt-bash
drwxr--r--   2 kolega  w4   2G Apr 20 21:37 nadzy-panowie
-rw-------   2 gebala  ki  10G Jul 10 14:88 przyjemny-egzamin-poradnik.pdf
```

W pierwszej kolumnie mamy opisane następujące pojęcia (od lewej do prawej):

- Pierwsza litera opisuje rodzaj bytu (`b` oznacza urządzenie blokowe, `s` gniazdko a `d` — katalog; myślnik oznacza zwykły plik).
- Trzy litery opisują flagi uprawnień do tego bytu dla klasy dostępu kolejno *owner*, *group*, *other*.

Jak widać do folderu `projekt-bash` pełen dostęp ma właściciel `konrad`, grupa `rzs` ma dostęp do odczytu i zapisu, natomiast inni użytkownicy nie mają żadnych uprawnień. Przebrnięcie przez folder `nadzy-panowie`, bądź plik `przyjemny-egzamin-poradnik.pdf`, zostawiam jako ćwiczenie.

13.5.1 Grupy

Również można zarządzać grupami w Linuksie. Grupy stanowią ułatwienie dla zarządzania uprawnieniami dla większej rzeszy ludzi niż sam właściciel pliku. By utworzyć grupę, można wywołać

```
# groupadd ppt
# groupadd w4
```

a potem dodać użytkowników za pomocą polecenia:

```
# gpasswd --add rmm ppt
# gpasswd --add kolega w4
```

Wtedy, mając taką grupę, możemy przypisać ją do pliku, bądź folderu, przydzielając tę samą grupę podkatalogom i plikom zawartym w tym folderze:

```
# chgrp ppt notatki.pdf
# chgrp -R ppt folder-rzs
```

czy też, bardziej kompleksowo, gdy chcemy zmienić właściciela i grupę:

```
# chmod konrad:ppt notatki.pdf
# chmod -R kolega:w4 nadzy-panowie
```

Można również usunąć użytkownika z danej grupy

```
# gpasswd --delete rmm ppt
```

oraz w ogólności usunąć całą grupę

```
# groupdel w4
```

Inne niewymienione funkcjonalności grup można poznać korzystając z podręcznika użytkownika.

13.6 Deskryptory plików

Dostęp do plików poprzez jądro systemu odbywa się na poziomie deskryptora. Deskryptor to przypisana nieujemna liczba całkowita danemu pliku dla danego procesu. Deskryptory są używane jako „haczyk” do pliku, do którego chcemy się odwoływać.

Domyślnie każdy program, bez manipulacji na deskryptorach, posiada trzy otwarte pliki — standardowe wejście (z deskryptorem 0), standardowe wyjście (z deskryptorem 1), standardowe wyjście błędów (z deskryptorem 2). By uzyskać deskryptor dla pliku nieotwartego, korzystamy z wywołania funkcji `open`.

Więcej szczegółów zostanie podane w przeglądzie wybranych funkcji dot. wejścia-wyjścia.

13.7 Wywołania systemowe

Linux oczywiście udostępnia interfejs wywołań systemowych, dzięki których można wykonywać manipulacje na plikach.

Na ogół wywołania systemowe są wolniejsze niż te, które są buforowane — wynika to z prymitywności wywołania; wczytane, bądź zapisane dane nie są ładowane do pamięci pośredniej, co wyraźnie spowalnia działanie w wypadku częstego dostępu przez takie funkcje, w przeciwieństwie do wysokopoziomowych bibliotek, jaką jest m. in. biblioteka standardowa.

Wywołania systemowe zostaną wymienione jako najbardziej potrzebne i najczęściej wykorzystywane w tabeli 13.4. Wszystkie wymienione funkcje można dołączyć z pliku `unistd.h`.

13.8 Potoki

Potoki są implementacją prostego mechanizmu komunikacji między procesami, a w szczególności między wątkami, polegającego na utworzeniu dwukierunkowego strumienia danych. O potoku możemy myśleć jako wirtualny tymczasowy plik, który istnieje tylko w pamięci podręcznej komputera i który jest czytany przez różne procesy. Potok, w kontekście powłoki tekstowej, jest to przekierowanie standardowego wyjścia programu na standardowe wejście innego, kolejnego programu będącego w potoku. Głównie takie potoki zapisuje się tak:

```
$ ls | less
```

Powyżej przekierowujemy wyjście, będące listą plików w katalogu, do aplikacji służącej do podglądu plików tekstowych.

Potoki stały się tematem głównym dla przetwarzania potokowego. Przetwarzanie potokowe to sposób użycia kilku programów o wyspecjalizowanej funkcji, które dają nam jakiś konkretny rezultat. Filozofia Uniksa wraz z przetwarzaniem potokowym daje niebywałą elastyczność w przetwarzaniu informacji. Przykładowo możemy przekierować wejście programu czytającego plik źródłowy C do programu, który usunie nam komentarze z pliku, a następnie ten rezultat przekierować do serwisu hostingu plików źródłowych. Powyższa operacja może wyglądać tak

```
$ cat zakazane.c | sed 's,/\*\*,,g;s,\*/,,g;s,/\*,,g;s,//,,g' | pastebin-cl
```

co jest bardzo wygodne. Również ciekawym pomysłem wydaje się pobieranie skompresowanego archiwum i wypakowywanie go w locie:

```
$ wget -O - http://moja.strona.pl/archiwum.tar.gz | tar -xvvf - -O ~/docelowy/folder
```

Głównie realizacja potoków odbywa się przez wywołania systemowe, o których będzie mowa w następnym rozdziale.

13.8.1 Zarządzanie potokami

Jądro Linuksa udostępnia narzędzia do tworzenia potoków w dowolnym kierunku. Mogą być to nawet potoki z obydwu kierunków.

Potoki często wykorzystuje się w programowaniu, gdy stronami potoku są pary procesów rodzica i dziecka, często też temu towarzyszy wykonanie innego programu przez jedną z funkcji `exec`.

Narzędzia ku tworzeniu potoków i zarządzania nimi są dość skąpe, ale wyczerpujące zastosowania. W dość minimalistycznej tabeli 13.5 zostaną wymienione wywołania dot. zarządzania potokami. Potoki traktuje się jak pliki, wobec tego można korzystać z funkcji wymienionych w tabeli 13.4 w ramach deskryptorów potoku.

Tablica 13.4: Przegląd wywołań systemowych dot. zarządzania plikami

`int open(const char *pathname, int flags);`
`int open(const char *pathname,`
`int flags, mode_t mode);`
	Otwiera, bądź tworzy, plik z wybranymi flagami, trybem otwarcia. Zwraca deskryptor, jeśli operacja się powiodła, `-1` w razie niepowodzenia.

`ssize_t read(int fd, void *buf,` `size_t count);`	Wczytuje dane z otwartego pliku do bufora, co najwyżej `count` bajtów. Zwraca ilość wczytanych bajtów do bufora, `-1` jeśli trafiono na koniec pliku.
`ssize_t write(int fd, void *buf,` `size_t count);`	Zapisuje `count` bajtów do otwartego pliku z bufora. Zwraca ilość zapisanych bajtów do bufora, `-1` jeśli wystąpił błąd.
`int close(int fd);`	Zamyka dany plik i czyni deskryptor bezużytecznym. Zwraca `0` podczas powodzenia, w przeciwnym wypadku `-1`.
`int dup(int oldfd);` `int dup2(int oldfd, int newfd);`	Tworzy kopię deskryptora pliku. W wypadku `dup` wybierana jest najniższa nieujemna liczba, która jest niezajęta, natomiast `dup2` ma określony jawnie nowy deskryptor. Zwraca wartość nowego deskryptora w podczas sukcesu, w przeciwnym wypadku `-1`.
`int setuid(uid_t uid);`	Jeśli proces wykonujący tę funkcję ma włączoną flagę `s` dla wybranego użytkownika oraz `uid` odpowiada temu użytkownikowi, to zmienia `uid` dla tego procesu. Wykorzystywane często do eskalacji uprawnień (por. UAC w Windowsie). Zwraca `0` jeśli został zmieniony `uid`, w przeciwnym wypadku `-1`.

Tablica 13.5: Wywołania systemowe dot. zarządzania potokami

`int pipe(int pipefd[2]);` `int pipe2(int pipefd[2], int flags);`	Tworzy parę deskryptorów opisującą potok. Domyślnie potok jest dwukierunkowy. By uczynić go jednokierunkowym, należy zamknąć w procesie potomnym i rodzicielskim jedną z dwóch stron potoku. Również można zmienić zachowanie funkcji poprzez flagi (zob. `man 2 pipe`). Zwraca `0` jeśli udało się stworzyć potok, to znaczy, utworzyć parę deskryptorów, gdzie `pipefd[0]` to wejście potoku, a `pipefd[1]` wyjście potoku, w wypadku niepowodzenia zwraca `-1`.

13.8.2 Potoki nazwane

Potoki nazwane to jawne pliki będące odpowiedzią na problem tworzenia kilku potoków jednocześnie. Każdy potok nazwany ma swoją nazwę i byt w systemie plików. Potok nazwany od potoku różni się tym, że potok nazwany jest bytem systemu plików, a potok jest anonimowym medium komunikacyjnym między procesami. Tabela **??** opisuje wywołania systemowe dot. potoków nazwanych. Również istnieje program `mkfifo`, który tworzy, korzystając z tego wywołania, potok nazwany z linii poleceń.

Tablica 13.6: Wywołania systemowe dot. potoków nazwanych

`int mkfifo(const char *pathname,` `mode_t mode);`	Tworzy potok nazwany, podawszy wybraną ścieżkę utworzenia pliku wraz z uprawnieniami do tego pliku. Zwraca nieujemny deskryptor, jeżeli uda się utworzyć potok nazwany, w przeciwnym wypadku zwraca `-1`.

13.9 Gniazdka

Gniazdko jest plikiem odzwierciedlającym medium komunikacyjne między procesem a procesem innego komputera będącego w jednej sieci komputerowej. Może być to również jeden i ten sam komputer, jeśli mówimy o zaawansowanej technice komunikacji między procesami (por. serwer graficzny X, bądź serwer dźwiękowy PulseAudio); odbywa się to dzięki tzw. *loopback*, czyli pętli zwrotnej.

13.9.1 Realizacja gniazdek w systemach uniksopodobnych

Gniazdo jest plikiem, z którym komunikacja odbywa się za pośrednictwem modułu odpowiadającego za warstwę transportu. Generalnie tworząc gniazdo, tworzymy je na poziomie abstrakcji warstwy transportu — protokoły, które będą obowiązywać między procesami, muszą być ustalone przez programistę implementującego komunikację między procesami.

Gniazdka w Linuksie, mimo architektury klient-serwer, albo P2P, nie są konkretnie spolaryzowane; gniazdko klienta i serwera w protokole TCP nadal jest tym samym gniazdkiem, lecz inaczej skonfigurowanym.

Tablica 13.7: Wywołania systemowe dot. gniazdek

`int socket(int socket_family, int socket_type, int protocol);`	Tworzy gniazdko, podawszy rodzinę gniazdka (zazwyczaj jest to `AF_INET`), rodzaj gniazdka (dla TCP wybieramy `SOCK_STREAM`, zaś dla UDP — `SOCK_DGRAM`) oraz rodzaj protokołu (zazwyczaj ustawia się `0`). Zwraca deskryptor pliku, który reprezentuje gniazdko, w wypadku niepowodzenia zwraca `-1`.
`struct hostent *gethostbyname( const char *hostname);`	Poszukuje nazwy w serwerze nazw (DNS). Jeśli uda się, funkcja zwraca strukturę zawierającą informację na temat adresu IP danego komputera, w przeciwnym wypadku zwraca wskaźnik zerowy.
`int connect(int sockfd, struct sockaddr_in *addr, socklen_t addrlen);`	Łączy się z hostem, podawszy deskryptor utworzonego uprzednio gniazdka, adres docelowy (konfiguracja odbywa się przez wyzerowanie struktury i skopiowania danych, dokładniej pola `h_addr` struktury uzyskanej z funkcji `gethostbyname`, jej wielkość jest podana w polu `h_length`) oraz wielkość struktury opisującej adres docelowy, zwyczajnie podaje się `sizeof(struct sockaddr_in)`. Zwraca `0`, gdy uzyskano połączenie, w przeciwnym wypadku `-1`.
`int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);`	Tworzy gniazdko serwerowe, w wypadku TCP. Analogiczne jak funkcja `connect`, jednakże konfiguracja obywa się tylko za pomocą ustawienia pola dot. portu; wykorzystuje się do tego funkcję `htons()`. Zwraca `0`, jeśli powiodło się, w przeciwnym wypadku `-1`.

W tabeli **??** są podane wywołania systemowe, które pozwalają na zarządzanie gniazdkami. Oczywiście gniazda są w ogólności plikami, stąd można korzystać z nich w funkcjach wymienionych w tabeli 13.4.

13.9.2 Selektory

Zauważono, że wykorzystywanie wielu wątków do obsługi wielu połączeń po stronie serwera jest bardzo niewydajnym rozwiązaniem — proszę sobie wyobrazić serwery Facebooka mające uruchomione około miliarda wątków w punkcie szczytu. Wobec tego postanowiono uczynić model zdarzeniowy dla gniazdek. Tworzy się tzw. *event-loop*, który oczekuje na zmianę stanu gniazdek, jeśli chodzi o możność czytania, bądź zapisywania danych do gniazdek. Mechanizm, który to udostępnia, nazywa się selektorem i odbywa się przez odpowiednie wywołania systemowe, które są opisane w tabeli **??**. Oczywiście selektory można wykorzystywać nie tylko z samymi gniazdkami, można również tworzyć selektory z innymi plikami, które zachowują się w podobny sposób co gniazdka — przykładowo standardowe wejście nie zawsze zawiera dane, więc można na nie oczekiwać.

Tablica 13.8: Wywołania systemowe dot. selektorów

`void FD_CLR(int fd, fd_set *set);`
`int FD_ISSET(int fd, fd_set *set);`
`void FD_SET(int fd, fd_set *set);`
`void FD_ZERO(fd_set *set);`
	Zestaw makr służących do ustalenia zbioru zainteresowań (ang. *intrest set*). Zbiór zainteresowań to zbiór deskryptorów, które chcemy obserwować. Odpowiednie funkcje, które są intuicyjne z nazwy, manipulują na zbiorze zainteresowań, bądź sprawdzają czy dany element w zbiorze zainteresowań zmienił swój stan. Zbiór zainteresowań może być zmieniany w każdym punkcie działania programu.

`int select(int nfds,`
`fd_set *readfds,`
`fd_set *writefds,`
`fd_set *exceptfds,`
`struct timeval *timeout);`
	Oczekuje na nadchodzącą zmianę w odpowiednich zbiorach zainteresowań (niektóre mogą być `NULL`), podawszy za `nfds` największy deskryptor plików spośród trzech zbiorów plus jeden oraz czas oczekiwania na zmianę (może być `NULL`), oraz odblokowuje wątek, jeśli w którymś ze zbiorów zainteresowań nastąpiła zmiana. Zwraca ilość ustawionych elementów w każdym ze zbiorów zainteresowań, w wypadku błędu zwraca `-1`. Funkcja `FD_ISSET` pozwala na sprawdzenie, który z elementów zbioru zainteresowań opisanego deskryptorem zmienił swój stan (np. możliwe jest wczytanie danych od niego).

Rozdział 14

Zarządzanie pamięcią

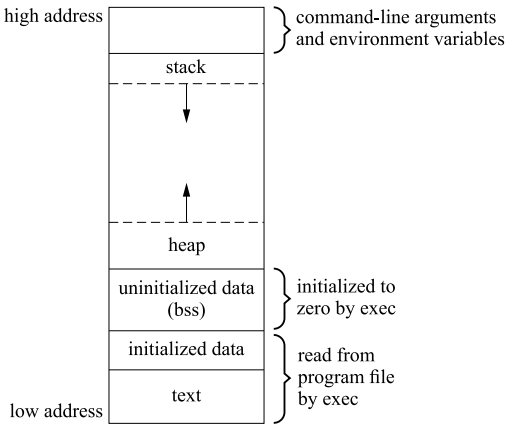
14.1 Układ pamięci programu

Pamięć większości programów, a szczególnie tych napisanych w języku C, składa się pięciu segmentów:

- Segment tekstu
- Segment zainicjalizowanych danych
- Segment niezainicjalizowanych danych
- Sterta (heap)
- Stos (stack)

W tej sekcji przyjrzymy się każdemu z wyżej wymienionych segmentów i pokrótce je omówimy

Rysunek 14.1: Schemat układu pamięci programu



14.1.1 Tekst

Znajdujący się na dole segment tekstu, zwany również segmentem kodu, zawiera w sobie wykonywalne instrukcje. Jako, że segment ten jest zazwyczaj typu 'read-only', często jest on współdzielony między wieloma procesami. Dzięki temu w pamięci musi istnieć tylko jedna kopia kodu dla procesów działających wiele razy i w dużych ilościach.

14.1.2 Zainicjalizowane dane

W tym segmencie, zwanym również, po prostu, segmentem danych, zawarte są wszystkie zmienne globalne i statyczne, których wartości definiuje programista. Segment ten znajduje się w wirtualnej przestrzeni adresowej i jest inicjalizowany przez wywołanie systemowe `exec` na wartości zawarte w kodzie programu.

14.1.3 Niezaincjalizowane dane

Segment ten, znany również jako 'BSS' (od frazy *block started by symbol*), znajduje się zaraz za segmentem danych. Zawiera on w sobie wszelkie zmienne globalne i statyczne, które (w przeciwieństwie do segmentu danych) nie zostały explicite zaincjalizowane przez programistę. Wszystkie zawarte w nim dane są ustawiane na wartość `arytemetyczne zero` przy starcie programu.

14.1.4 Sterta

Segment sterty zaczyna się zaraz za 'BSS' i rośnie w górę. To w nim, zazwyczaj, ma miejsce dynamiczne alokowanie pamięci. Sterta powiększa się gdy używana przez programistę metoda alokacji pamięci nie będzie w stanie znaleźć miejsca na nowe dane (O dostępnych metodach później). Segment ten jest współdzielony przez wszystkie biblioteki oraz dynamicznie ładowane moduły naszego procesu.

14.1.5 Stos

Segment stosu znajduje się na przeciwko sterty i rośnie w dół (w kierunku adresu `zero`). Dawniej, gdy szczyty tych dwóch regionów pamięci spotkały się, programowi, po prostu, kończyła się wolna pamięć. Dziś zastosowane są różne triki, takie jak wirtualna przestrzeń pamięci dzięki którym segmenty te mogą znajdować się w dowolnych miejscach pamięci, więc ryzyko "zderzenia" jest niewielkie. Z uwagi na swoją strukturę alokacja na stosie jest zdecydowanie szybsza tej na stercie. Dzieje się tak, ponieważ w przypadku stosu nie musimy "szukać" wolnego miejsca w pamięci, obniżamy tylko wskaźnik szczytu.

14.2 Alokacja pamięci

Do dynamicznej alokacji pamięci programista ma kilka narządzie. Niektóre z nich bardziej, a niektóre mniej niskopoziomowe. W tej sekcji omówimy podstawowe z nich.

14.2.1 Wywołania `brk`, `sbrk` oraz `mmap`

Wywołanie systemowe `brk` jest najbardziej niskopoziomowym sposobem alokacji pamięci na stercie z wymienionych w tej sekcji. Bierze ono, jako argument, adres w wirtualnej przestrzeni pamięci który ma się stać

nowych szczytem sterty. Dzieje się tego tylko wtedy, gdy system operacyjny uzna żądany adres za rozsądny. W przypadku sukcesu wywołania, zwraca ono wartość 0, inaczej -1.
Kolejnym wywołaniem systemowym jest `sbrk`, które można uznać za swego rodzaju nakładkę na `brk`. Tym razem jako argument nie podajemy konkretnego adresu ale ilość bajtów o jaką chcemy podnieść szczyt sterty. Wywołanie to można również użyć do znalezienia aktualnego położenia szczytu. W przypadku sukcesu wywołania, zwraca ono wskaźnik na poprzedni szczyt, inaczej -1 (jako `void*`).
Ostatnią metodą dynamicznej alokacji pamięci przez wywołania systemowe jaką opiszemy jest mapowanie. Technika te jest głównie wykorzystywana do ładowania całych plików do RAMu, co umożliwia szybki dostęp do ich zawartości, jednak jeśli mapowanie jest anonimowe, jego rezultatem będzie alokacja podanej w argumentach ilości wyzerowanej pamięci. Należy pamiętać, że pamięć jest alokowana w wirtualnej przestrzeni pamięci programu, a nie stricte na stercie. W niektórych implementacjach, funkcja `malloc` (O której później) korzysta z tego wywołania przy alokacji większej ilości pamięci.

14.2.2 Funkcja `malloc`

Funkcja `malloc` próbuje zaalokować podaną jako argument ilość bajtów pamięci na stercie. Do tego wykorzystuje wywołania systemowy `sbrk` oraz czasem `mmap`. Zazwyczaj stosowanie jest tylko to pierwsze jako, że jest bardzo szybkie oraz zwraca gotowy adres początku zaalokowanej pamięci, który później zwraca nam `malloc`. W przypadku żądań o większą ilość bajtów (Zazwyczaj mowa tu o liczbach przekraczających cztery tysiące), niektóre implementacje tej funkcji użyją anonimowego mapowania.
Należy pamiętać, że praca funkcji nie kończy się na użyciu wywołania systemowego. Chociażby dlatego, że czasem nie jest ono wymagane. Typowa implementacja `malloc` używa również algorytmów zapobiegających fragmentacji sterty, tzn umieszczania nowych danych w miejscach, które wcześniej zostały zaalokowane, ale były już zwolnione.
Należy również pamiętać, że każda dynamicznie alokowana pamięć powinna być zwalniana, gdy nasz program przestaje jej używać. Nie dzieje się to automatycznie (Co z punktu widzenia szybkości działania programu jest wielką zaletą).

Rozdział 15

Pliki wykonywalne ELF

Scoia'tael!

(Okrzyk bojowy elfów)

Plik wykonywalny to taki, który może być uruchomiony bezpośrednio w środowisku systemu operacyjnego. Najczęściej zawiera on binarną reprezentację instrukcji konkretnego procesora. Mogą się w nim znajdować również wywołania systemowe, co czyni takie pliki nie tylko specyficznymi dla danego typu procesora, ale też dla systemu operacyjnego. Istnieją również pliki wykonywalne w formie pośredniej, które do uruchomienia wymagają odpowiedniego interpretera lub maszyny wirtualnej - takie pliki, w dużej części przypadków, mogą być już uruchamiane na różnych rodzajach systemów.

W naszym skrypcie skupimy się na plikach wykonywalnych systemu Linux. Pierwszymi historycznie plikami były `a.out`. Jest to skrót od *assembler output*. Była to nazwa pliku wyjściowego generowanego przez assembler Kena Thompsona dla PDP-7. Format ten był wykorzystywany w starszych systemach typu Unix-like. Pliki `a.out` funkcjonowały jako domyślna nazwa plików wykonywalnych generowanych przez kompilatory `gcc` oraz `g++` (jednak, o czym zapewne Czytelnik doskonale wie, ten format nie jest już używany). W systemach z rodziny BSD format ten został początkowo zastąpiony przez COFF (*Common Object File Format*), a następnie przez pliki typu ELF (*Executable Linking Format*). W systemach typu Linux format `a.out` był wykorzystywany do wersji 1.2, a później został zastąpiony formatem ELF (głównie dla bibliotek współdzielonych). W przypadku systemu Windows mamy do czynienia z plikami wykonywalnymi będącymi rozszerzeniem plików COFF, o formacie PE (*Portable Executable*).

Skupmy się teraz na temacie rozdziału, czyli plikach ELF. Ten format plików wykonywalnych powstał w *AT&T Bell Laboratories* dla systemu UNIX System V i, jak zostało wspomiane w jednym z poprzednich akapitów, zastąpił on format `a.out`. Powodem takiej zmiany była potrzeba stworzenia formatu pliku wykonywalnego z lepszym wsparciem dla bibliotek dynamicznych.

Plik ELF możemy podzielić na dwie zasadnicze części - nagłówek oraz sekcje. Nagłówek zawiera informacje o sposobie organizacji pliku i zawsze znajduje się na jego początku. Tutaj znajdziemy również tablicę nagłówkową programu, zawierającą informacje o tym, w jaki sposob system ma stworzyć obraz procesu, a także tablicę nagłówków sekcji, która przechowuje informacje o każdej sekcji - jej nazwie, rozmiarze itd. O sekcjach natomiast opowiemy sobie więcej w pierwszej części tego rozdziału.

15.1 Sekcje pliku wykonywalnego

Plik wykonywalny ELF może zostać podzielony na dowolną liczbę sekcji o dowolnej wielkości każda. Każda sekcja ma określone cechy - ochronę oraz adres wirtaulny. Każda z sekcji może zostać również oznaczona jako taka, która nie jest ładowana do pamięci. Zlokalizowanie odpowiedniej sekcji w pliku jest możliwe poprzez wykorzystanie tablicy nagłówków sekcji. Każda sekcja w pliku obiektowym posiada dokładnie jeden nagłówek. Każda sekcja zajmuje jedną ciągłą (możliwe, że pustą) sekwencję bajtów w pliku. Sekcje w pliku nie mogą się nakładać. Żaden bajt nie może być w więcej niż jednej sekcji. Plik obiektowy może posiadać przestrzeń nieaktywną. Różne nagłówki i sekcje mogą nie pokrywać każdego bajtu w pliku obiektowym. Zawartość takich danych nieaktywnych jest niezdefiniowana. Nagłówek sekcji wygląda w następujacy sposób:

```
typedef struct {
      Elf32_Word        sh_name;
      Elf32_Word        sh_type;
      Elf32_Word        sh_flags;
      Elf32_Addr        sh_addr;
      Elf32_Off                sh_offset;
      Elf32_Word        sh_size;
      Elf32_Word        sh_link;
      Elf32_Word        sh_info;
      Elf32_Word        sh_addralign;
      Elf32_Word        sh_entsize;
  } Elf32_Shdr;
```

15.2 Plik relokowalny

Plik relokowalny jest rodzajem pliku obiektowego generowanego przez kompilator lub asembler podczas kompilacji pliku z kodem źródłowym. Taki plik może zostać również utworzony przez linker (konsolidator) w wyniku połączenia kilku plików relokowalnych. Takie pliki są przeznaczone do późniejszego wykorzystania przez linker w celu otrzymania pliku relokowalnego lub biblioteki dynamicznej. W systemie Unix rozszerzenie takiego pliku to `*.o`.

15.3 Biblioteki statyczne

Biblioteka statyczna (*static library*) jest to rodzaj biblioteki funkcji, która łączona jest z programem w momencie konsolidacji. W przypadku systemu Unix są to pliki o roszerzeniach `*.a` lub `*.o`.

15.4 Biblioteki dynamiczne

Biblioteka dynamiczna (*dynamic library*) to rodzaj biblioteki, która jest łączona z programem dopiero w momencie jego wykonania. Dane z takich bibliotek mogą być współdzielone jednocześnie przez różne programy. Są one ładowane do pamięci tylko raz, nawet jeżeli są jednocześnie współużytkowane.

Rozdział 16

System plików FAT

Następny przystanek: FAT. Koniec trasy!

(Pan z głośników w tramwaju)

Fabryka Automatów Tokarskich, która ma swoją siedzibę przy ul. Grabiszyńskiej we Wrocławiu, powstała w roku... O, przepraszam! To nie ta strona na Wikipedii... </żart>

Po całym semestrze kursu Architektura Komputerów i Systemy Operacyjne Czytelnik na pewno nie jeden raz słyszał pojęcie „system plików". W dość dużym skrócie (ale w zakresie, który nam w zupełności w tym momencie wystarczy) możemy tak nazwać sposób organizacji danych na nośniku pamięci, który umożliwia przechowywanie oraz zarządzanie plikami w sposób przyjazny dla użytkownika. Instalując dowolnego Linuksa spotkaliśmy się najpewniej z systemem plików z rodziny ext (obecnie najpopularniejszym jest ext4). W Windowsie natrafimy na takie systemy jak NTFS czy FAT, któremu poświęcony jest ten rozdział skryptu. Osoby, które trochę bardziej interesują się tematem natknęły się pewnie jeszcze na systemy takie jak minix, XFS, JFS...

FAT czyli *File Allocation Table*, to system plików powstały pod koniec lat 70. ubiegłego stulecia. Pierwotnie FAT był przeznaczony do użytku na dyskietkach, ale w późniejszym okresie został przystosowany i powszechnie stosowany na dyskach twardych komputerów. Swoje zastosowanie znalazł on najpierw w systemie DOS firmy Microsoft, a później - w Windowsie. Pierwotnie FAT był przeznaczony do użytku na dyskietkach, ale w późniejszym okresie został przystosowany i powszechnie stosowany na dyskach twardych komputerów.

Najważniejszym elementem systemu FAT jest tablica alokacji (stąd nazwa), w której przechowywane są informacje o rozmieszczeniu plików na partycji dyskowej. System jest podzielony na tak zwane "klastry", które są podstawową jednostką wykorzystywaną do alokacji. Tablica alokacji przechowuje na sobie informacje, który klaster jest wykorzystywany przez który plik.

Możemy wyróżnić kilka rodzajów opisywanego systemu - są nimi: FAT12, FAT16, FAT32 oraz exFAT (FAT64). Podstawowa różnica między nimi to ilość bitów przeznaczonych na zapisanie klastra - na przykład, dla systemu FAT12, mamy przeznaczone 12 bitów na klaster, co oznacza, gdy sobie to szybko policzymy, że na 3 bajtach możemy zapisać 2 klastry. W wypadku systemu FAT16 - 2 bajty to 1 klaster, a gdy mówimy o FAT32 - 1 klaster to 4 bajty.

Skupmy się teraz na szczegółach budowy systemu FAT. Zanim przejdziemy do omówienia klastrów, tablicy alokacji oraz tablicy katalogu głównego, przyjrzyjmy się jeszcze części FATa znanej pod nazwą *boot sector*.

Znajdziemy tutaj podstawowe informacje takie jak: rodzaj FATa, nazwa partycji, ilość bajtów przeznaczonych na sektor, czy ilość sektorów w jednym klastrze. Wygląd *boot sectora* różni się w zależności od rodzaju systemu FAT. Informację o tym, co oznaczają poszczególne bajty możemy znaleźć w odpowiedniej dokumentacji. Nie będę jednak skupiał się w tym miejscu na tych rzeczach, ponieważ nie jest to w tym momencie coś, co byłoby nam w znaczący sposób potrzebne. Przejdźmy zatem dalej.

16.1 Klastry

O klastrach wspominałem już we wcześniejszych akapitach. W systemie FAT partycja podzielona jest na klastry, czyli jednostki alokacji pliku. Wszystkie klastry mają swoje "imię" (są numerowane) i są podzielone na sektory. Każdy z klastrów przechowuje na sobie jakiś plik (lub jego część, gdy rozmiar pliku przekracza całkowity rozmiar jednego klastra). Informację o tym, od którego klastra zaczyna się plik, znajdziemy w odpowiednim miejscu w tablicy katalogu głównego. Natomiast wskazania na kolejne klastry powiązane z konkretnym plikiem znajdziemy w tablicy alokacji plików (o tych dwóch rzeczach napiszę więcej w rozdziałach poświęconych odpowiednio tablicy alokacji i tablicy katalogu głównego.

Klaster, w zależności od rodzaju FATa, ma określony rozmiar. I tak - w przypadku systemu FAT12 rozmiar jednego klastra to od 512 B do 4 kB. Maksymalna liczba klastrów wynosi 4077 ($2^{12} - 19$), a maksymalna pojemność pliku, jak i całego woluminu, to w tym wypadku 16 MB. W systemie FAT16 mamy już 65 517 klastrów ($2^{16} - 19$), a rozmiar jednego takiego klastra to 32 kB. Maksymalny rozmiar woluminu i pliku to 2 GB. W FAT32 spotykamy się z plikiem o maksymalnym rozmiarze 4 GB, ale rozmiary woluminu mogą już dochodzić do 2 TB. Górne ograniczenie liczby klastrów to 268 435 445, a maksymalny rozmiar jednego klastra wynosi 256 kB. System exFAT umożliwa już utworzenie pliku i woluminu o rozmiarach 16 EiB (!), a maksymalna liczba klastrów to $2^{32} - 1$.

16.2 Tablica alokacji plików

Tablica alokacji plików to "serce" systemu FAT. Znajduje się zaraz za *boot sectorem*. To tutaj znajdują się informacje o tym, na jakich klastrach zapisany jest konkretny plik - to znaczy, że możemy tutaj odczytać numery poszczególnych klastrów (informację o miejscu, od którego należy zacząć czytać znajdziemy w tablicy katalogu głównego, ale o tym za chwilę). Wartości, które znajdziemy w tablicy alokacji, znowu zależą od rodzaju systemu FAT. I tak - w przypadku systemu FAT12 znajdziemy tam następujące wartości:

- `000` - wolne klastry
- `002-fef` - używane klastry
- `ff0-ff6` - klastry zarezerwowane
- `ff7` - błędny sektor
- `ff8-fff` - ostatni klaster

W systemie FAT16 wartości w tablicy alokacji odczytuje się w następujący sposób:

- `0000` - wolne klastry

- `0002-ffef` - używane klastry
- `fff0-fff6` - klastry zarezerwowane
- `fff7` - błędny sektor
- `fff8-ffff` - ostatni klaster

16.3 Tablica katalogu głównego

Ostatnia, ale nie mniej ważna część systemu FAT, to tablica katalogu głównego (*root directory*). Tutaj znajdziemy wszystkie potrzebne informacje na temat plików utworzonych w naszym systemie. Dla systemu FAT12 znaczenie poszczególnych bajtów wygląda następująco:

- 1-10 - nazwa pliku wraz z rozszerzeniem (8 bajtów jest przeznaczonych na nazwę, a 3 bajty na rozszerzenie)
- 11 - atrybuty pliku (wektor bitowy)
- 12-21 - zarezerwowane
- 22-23 - czas utworzenia (format: godzina/minuty/sekundy)
- 24-25 - data (format: rok/miesiąc/dzień)
- 26-27 - wskazanie na pierwszy klaster pliku
- 28-31 - rozmiar pliku w bajtach

Przy okazji omawiania *root directory* warto jeszcze wspomnieć o odzyskiwaniu plików w systemie FAT. Gdy usuwany jest plik, następuje wyzerowanie tablicy alokacji a także usunięcie z tablicy katalogu głównego informacji o pierwszej literze w nazwie usuniętego pliku (wszystkie pozostałem informacje pozostają bez zmian). W momencie, gdyby Czytelnik zechciał przywrócić taki plik, należy przywrócić pełną nazwę pliku wpisując do *root directory* odpowiedni kod ASCII pierwszej litery w nazwie pliku. Następnie, na podstawie informacji o numerze pierwszego klastra i o rozmiarze pliku, należy podjąć się ręcznego przywracania tablicy alokacji, co przy dużych rozmiarach plików jest zajęciem niezwykle żmudnym. Należy tutaj jednak zaznaczyć, że takie przywrócenie usuniętego pliku jest możliwe tylko wtedy, gdy nie mamy do czynienia z fragmentacją pliku. W momencie, gdy następuje znaczna fragmentacja, przywrócenia takiego pliku jest bardzo trudne lub wręcz niemożliwe!

Rozdział 17

Język C

Ostatnią naprawdę dobrą rzeczą napisaną w C była 9. Symfonia Schuberta

(Erwin Dietrich)

Język C powstał w Bell Labs w latach 1969–1973, jego twórcą był Dennis Ritchie. W roku 1973 udało się zaimplementować pierwszą wersję jądra Uniksa. Pięć lat później Brian Kernighan i Dennis Ritchie opublikowali dokumentację języka pt. *C Programming Language*. C stał się popularny poza Bell Labs po 1980 roku i stał się dominującym językiem do programowania systemów operacyjnych i aplikacji. Stąd poniższy rozdział będzie poświęcony językowi C i jego podstawowymi konstrukcjami. Autor zakłada, że Czytelnik miał już styczność z językiem C w sposób chociażby połowiczny. Główna treść tego rozdziału będzie się opierać bardziej na implementacyjnej naturze języka C i zachowania kompilatorów, głównie GCC.

17.1 Typy danych

Język C jest językiem o słabym[1] typowaniu statycznym. Głównie typy w języku C dzielimy na podstawowe i złożone.

17.1.1 Typy proste

Typami prostymi są zwyczajnie liczby całkowite, liczby zmiennoprzecinkowe, znaki, wartości boolowskie, które są po prostu liczbami całkowitymi, które interpretuje się następująco: jeśli liczba całkowita jest różna od zera, jest to prawda, w przeciwnym wypadku jest to fałsz. Wszelkie typy proste będą opisane w tabeli 17.1.

17.1.2 Typy złożone

Typami złożonymi nazywamy typy powstałe z innych typów złożonych, bądź prostych. W zależności od rodzaju typu złożonego, mamy różne zachowania towarzyszące rozmieszeniu danej zmiennej typu złożonego w pamięci. Informacje, które opisują dogłębnie wszystkie typy złożone, można znaleźć w tabeli **??**.

[1] Słabym, to nie znaczy złym.

Tablica 17.1: Typy proste dla kompilatora GCC i MinGW (32-bitowe)

Typ	**Opis**	**Wielkość**	**Przedział wartości**	
Liczby całkowite w kodzie uzupełnieniowym do dwóch				
`char`	Znak w kodzie ASCIII	1	-2^7	2^7-1
`unsigned char`	Znak w kodzie ASCIII bez znaku	1	0	2^8-1
`short int`	Liczba całkowita krótka	2	-2^{15}	$2^{15}-1$
`unsigned short int`	Liczba całkowita krótka bez znaku	2	0	$2^{16}-1$
`int`	Liczba całkowita	4	-2^{31}	$2^{31}-1$
`unsigned int`	Liczba całkowita bez znaku	4	0	$2^{32}-1$
`long int`	Liczba całkowita długa	4	-2^{31}	$2^{31}-1$
`unsigned long int`	Liczba całkowita długa bez znaku	4	0	$2^{32}-1$
`long long int`	Liczba całkowita bardzo długa	8	-2^{63}	$2^{63}-1$
`unsigned long long int`	Liczba całkowita bardzo długa bez znaku	8	0	$2^{64}-1$
Liczby zmiennoprzecinkowe w standardzie IEEE-457				
`float`	Liczba zmiennoprzecinkowa pojedynczej precyzji	4	2^{-126}	$2\left(1-\frac{1}{65536}\right)\cdot 2^{127}$
`double`	Liczba zmiennoprzecinkowa podwójnej precyzji	8	2^{-1022}	$2\left(1-\frac{1}{65536}\right)\cdot 2^{1023}$
`long double`	Liczba zmiennoprzecinkowa poczwórnej precyzji	10	2^{-16382}	$2\left(1-\frac{1}{65536}\right)\cdot 2^{16383}$
Rozszerzenia kompilatora				
`bool`	Typ boolowski	1	-2^7	2^7-1

Tablica 17.2: Typy złożone w języku C

Nazwa konstrukcji	Przykład	Opis
Tablica	`int array[10];`	Ciągła struktura danych, która przypomina ciągłą alokację n zmiennych o tym samym typie. W przykładzie rezerwujemy tablice dziesięcioelementową o elementach typu `int`. Kompilator w takim wypadku rezerwuje na stosie dokładnie ns bajtów, gdzie s to wielkość typu elementów tablicy.

Struktura	`struct address_t {`
	`  int port;`
	`  char hostname[64];`
	`};`
		Heterogeniczna struktura danych, która zawiera w sobie nazwane elementy o różnych typach. By odwołać się do elementu ze zmiennej, będącą instancją struktury, korzystamy z notacji `struktura.pole`, bądź `wskaznik_do_struktury->pole`. Kompilator ustala wielkość na podstawie różnych strategii; jedna z nich będzie opisana w tym rozdziale.

Unia	`union ipv4_t {`
	`  char nibble[4];`
	`  int address;`
	`};`
		Struktura danych pozwalająca na interpretację danej na różny sposób. Przykład pokazuje szybką konwersję zmiennej typu `int` na tablicę czterech bajtów i odwrotnie. Wielkość unii jest uzależniona od największego typu będącego członkiem unii. Każdy element unii zaczyna się w tym samym miejscu, tzn. każdy członek ma taki sam adres początkowy, lecz nie końcowy.

Pole bitowe	`struct flags_t {`
	`  unsigned carry : 1;`
	`  unsigned zero : 1;`
	`  unsigned overflow : 1;`
	`};`
		Struktura danych pozwalająca na łatwą manipulację danych, które są mniejsze niż jeden bajt. Pole bitowe jest wielkości wszystkich szerokości podanych po dwukropku, zaokrąglając do całych bajtów. Szerokość musi nie przekraczać typu danego pola w polu bitowym.

Wyliczenie	`enum state_t {OFF, STARTED, STOPPED, OFF};`	Typ wyliczeniowy. Każdy członek domyślnie wylicza się od zera i kolejno o jeden więcej, zgodnie z listą w deklaracji typu.
Wskaźnik	`int *ptr;` `int **mptr;`	Wskaźnik odwołuje się do innego miejsca w pamięci. By odwołać się do danej w innym miejscu w pamięci, korzystamy z operatora dereferencji (`*ptr = 8`). By ustawić wskaźnik na wybraną zmienną, przypisuje się do wskaźnika adres danej zmiennej (`ptr = &foo`). Wskaźniki po assemblacji są zamieniane na adresowanie pośrednie.

17.2 Funkcje

Funkcje są to podprogramy realizujące jedno zadanie, które zwracają jakiś wynik, bądź powodują jakieś efekty uboczne. Funkcje, w przypadku procesorów x86, są realizowane w postaci kodu oznaczonego etykietą, którego wykonuje się zgodnie z wybraną konwencją wywołania zależną od systemu.

W wypadku Linuksa, stosuje się wariant `cdecl`, który wygląda zgrubnie następująco. Opis będzie rozkładał się na wywołującego i wywoływanego.

1. Odłóż argumenty potrzebne dla tej funkcji na stos od prawej do lewej, biorąc listę argumentów funkcji. Podobnie dla koprocesora x87.
2. Wywołaj funkcję.
   a) Odłóż na stos stary wskaźnik ramki stosu.
   b) Zapisz aktualną pozycję stosu na wskaźnik ramki stosu
   c) Odłóż na stos rejestry, które muszą być nienaruszone według konwencji.
   d) Wykonaj kod funkcji.
   e) Gdy ma dojść do zwrócenia wartości funkcji, rezultat jest odkładany w rejestrze `eax`.
   f) Przywróć rejestry ze stosu, które muszą być nienaruszone według konwencji.
   g) Przywróć stary wskaźnik ramki stosu.
   h) Powróć do wywołującego.
3. Wyczyść stos z tymczasowych zmiennych funkcji wywołanej.

Istnieją również inne konwencje wywołania, jednakże na nasze potrzeby, taka wiedza jest wystarczająca.

17.3 Wyrównywanie struktur

C jest językiem, mimo wszelkich pozorów, niskopoziomowym. To znaczy, język C stanowi wysokopoziomową, łatwą do zrozumienia, „wersją assemblera".

Problem, który zostanie tu poruszony, ze względu właśnie assemblerowej natury języka C, jest alokacja struktury. Struktura, jak wiemy, jest heterogeniczną strukturą danych, która może zawierać dane o różnej wielkości. Problemem jest ustalić kolejność występowania członków struktury i osadzić ich na liniowym odcinku pamięci.

Naiwnym rozwiązaniem byłoby zarezerwowanie ciągłego odcinka pamięci dla każdego z członka struktury po kolei. Jednakże to rozwiązanie niesie kłopoty w postaci nieregularnego adresowania danych. Wygenerowanie assemblera dla danych o takim sposobie alokacji by było bardzo niewydajne, wymagałoby wyliczania za każdym razem adresu w nieregularny sposób. Również sprawia problem wyciąganie danych ze struktury. Można sobie wyobrazić, że chcemy załadować wartość pewnego członka struktury do rejestru, lecz ten członek jest mniejszy od rejestru. Wobec tego, co jest technicznie jak najbardziej możliwe, nie wczytamy dokładnie wartości tego członka. Wymagałoby to specjalnych masek bitowych, co jeszcze bardziej pogrąża wydajność tego rozwiązania.

Jest również inny sposób alokacji członków takiej struktury. Alokujemy członków, wyrównując ich wielkość do pewnej, ustalonej wielkości (o tym, do jakiej wartości się wyrównuje, zostanie wspomniane później). To podejście z angielskiego nazywamy *alignment*, czyli wyrównywanie.

Tablica 17.3: Tabela wyrównań typów dla GCC (x86, ARM)

Typ	Wyrównanie
`char`	1
`short`	2
`int`	4
`long`	4
`long long`	4
`float`	4
`double`	4
`long double`	4
dowolny wskaźnik	4
tablica	wyrównanie wg typu elementu
struktura	wyrównanie wg największego wyrównania w strukturze

17.3.1 Algorytm wyrównywania

W C stosuje się wyrównywanie struktur, które jest uzależnione od wielkości typów.

Definicja. Adres a jest wyrównany do n bajtów, jeśli a jest wielokrotnością n bajtów, gdzie n jest potęgą dwójki.

Definicja. Struktura jest wyrównana, jeśli przy dostępie do danej o wyrównaniu do n bajtów wykorzystujemy adres a wyrównany do n bajtów. W przeciwnym wypadku struktura jest niewyrównana.

Tabela typów, które muszą być wyrównane do n bajtów dla ARM GCC i GCC (32-bitowe), znajduje się jako 17.3.

Ogólnie zasada obliczenia odpowiedniego wypełnienia dla każdego z pól struktury odbywa się następująco. Przesunięciem (ang. *offset*) będziemy nazywać odległość (w bajtach) od początku struktury do pola, dla którego obliczamy offset, wraz z poprzednimi wypełnieniami, jeśli były już wyliczone. Dopasowaniem (ang. *align*) będziemy nazywać jedną z wartości opisanej w tabeli 17.3. Algorytm wygląda następująco:

- Jeśli pole struktury o danym typie ma przesunięcie będące wielokrotnością dopasowania, to nie wymagane jest żadne wypełnienie.
- Jeśli pole struktury o danym typie nie ma przesunięcia będącego wielokrotnością dopasowania, to dodajemy tyle bajtów wypełnienia przed tym polem, by przesunięcie było wielokrotnością dopasowania (dodajemy *padding*).

Na końcu wybiera się element o największym dopasowaniu i dodaje się wypełnienie na końcu struktury, by jej końcowy rozmiar był równy wielokrotności tego największego dopasowania. By uzmysłowić działanie tego algorytmu, możemy wykonać

Zadanie 1. Rozważamy poniższą deklarację struktury w C.

```
typedef struct {
    long a;
```

```
    char b;
    void *c;
    short d[2][8];
} T;
```

Ile bajtów zajmuje tablica `T t[10]` w architekturze ARM (32-bitowej)? Odpowiedź uzasadnij.

Rozwiązanie.
Algorytm wyrównywania będzie pokazywany krok po kroku. Dla każdego typu zastanawiamy się, czy przesunięcie jest wielokrotnością dopasowania dla wybranego typu. Dopasowania są wymienione w tabeli 17.3.

Weźmy na początek pole `long a`. Powyższe pole ma przesunięcie równe $\delta_a = 0$. To pole jest trywialnie dopasowane. Weźmy zatem `char b`. Pole ma przesunięcie $\delta_b = 4$. To jest wielokrotność jedynki, więc żadne wypełnienie nie jest potrzebne.

Następne pole `void *c` jest wskaźnikiem. Przesunięcie wynosi $\delta_c = 5$. Nie jest to wielokrotność czwórki, dlatego przed polem dodamy wypełnienie o wielkości trzech bajtów.

```
typedef struct {
    long a;
    char b;
    char pad1[3]; // padding dla pola c
    void *c;
    short d[2][8];
} T;
```

Patrzymy na pole `short d[2][8]`. Widzimy, że to jest tablica dwuwymiarowa, jednakże ona jest równoważna (dla algorytmu) tablicy jednowymiarowej szesnastoelementowej. Mamy więc przesunięcie $\delta_d = 5 + 3 = 8$. Osiem jest wielokrotnością dwójki, stąd padding nie jest potrzebny.

Największym dopasowaniem w tej strukturze było 4 (ze względu na pole `long a`). Wobec tego mamy rozmiar całej struktury bez końcowego wypełnienia równy $4 + 1 + 3 + 4 + 16 = 28$. Zauważamy, że 28 jest wielokrotnością 4, wobec tego nasza struktura została wyrównana bez wypełniania końca struktury i jej rozmiar wynosi 28 bajtów.

Stąd możemy powiedzieć, że tablica ma rozmiar 280 bajtów.

Bibliografia

[1] Marek Andrzej Perkowski. *Multi-level logic minimization*. URL: http://web.cecs.pdx.edu/~mperkows/=FSM/finite-sm/node17.html.

[2] *Transistor count*. Grud. 2017. URL: http://en.wikipedia.org/wiki/Transistor_count#Logic_functions.