Model komputera

piątek, 29 marca 2024 16:46

Komputer z programem przechowywanym: Komputery służą do liczenia, które realizuje się uruchamiając programy

I. Program - ustalony zapis algorytmu, który można wykonać, **WAŻNE**! **Wykonuje operacje na jakichś danych.**

W tej chwili głównym problemem jest nie szybkość wykonywania działań, ale dostarczania danych.

II. Programowanie imperatywne - **sekwencja (ciąg)** rozkazów czyli instrukcji (dwa terminy) , one zmieniają stan programu (pamięć + rejestry procesora). **Rozkazy wykonują się po kolei!**

III. Maszyna Turinga - abstrakcja:

a) Taśma nieskończenie długa
b) Głowica
c) Stany maszyny
d) Rozkazy maszyny
"Program jest wykonywany instrukcja po instrukcji na takiej taśmie, po tej taśmie porusza się głowica i robi coś z jej polami "

Jak to wygląda naprawdę?

a) Taśma modeluje pamięć,
b) głowica modeluje procesor, który porusza się po pamięci, potrafi dane odczytywać , zapisywać i modyfikować
c) Stany maszyny to zawartość rejestrów procesora, które są niewielką pamięcią, można w niej coś przechowywać
Rozkazy???

Komputer zawiera jeszcze kluczowy element łączący procesor z pamięcią zwany **magistralą.**
Na magistrali są 3 rodzaje informacji:

1. adres komórki pamięci
2. dane przesyłane
3. dodatkowe sygnały (chcemy pisanie czy czytanie)

IV. Pamięć to tablica komórek pamięci, elementów o jednakowym rozmiarze (zazwyczaj bajtów). Te bajty możemy adresować. Minimalny rozmiar komórki pamięci w procesorze nazywa się **rozdzielczością adresowania.** Fizycznie pamięć jest ograniczona, ale w prostych problemach można traktować ją jako nieograniczoną.

Procesor składa się z:

a) Jednostka wykonująca działania (dodawanie , odejmowanie, mnożenie, dzielenie, operacje logiczne)
b) Jednostka sterująca (najbardziej skomplikowana ze względu na sposób działania)
c) Rejestry procesora
d) Magistrala wejścia/wyjścia*, jakaś jej forma jest potrzebna, bo inaczej maszyna liczyła by sama sobie

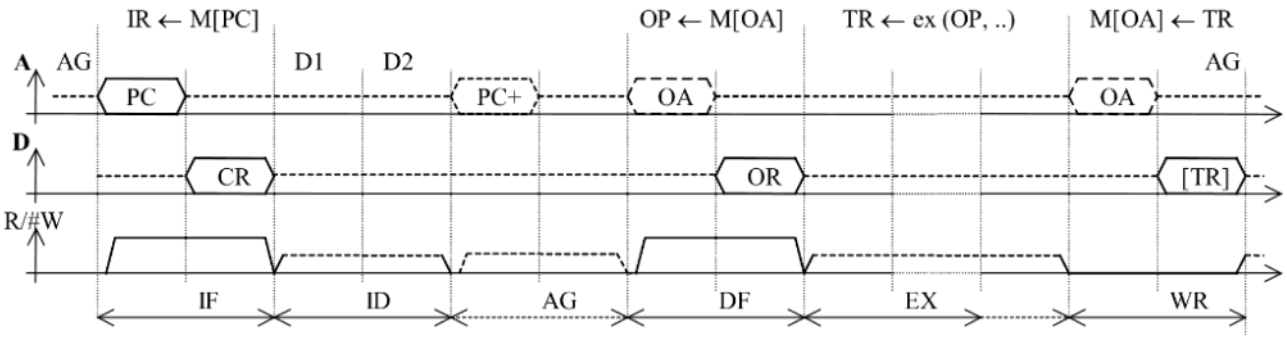
Pominięty prosty układ, który pokazał bez większego kontekstu.

Co robi procesor?

piątek, 29 marca 2024 18:45

Kiedy procesor odczytuje z pamięci poszczególne dane to z definicji robi to sekwencyjnie, odczytuje i na podstawie tego co odczytał oraz stanu wewnętrznego, zawartości rejestrów, to odczytane wartości z pamięci są traktowane albo jako kody operacji albo jako dane.
Wspomina o pętlach, ale też tak delikatnie trochę bez większego kontekstu

Cykl Rozkazowy (CISC: Complex Instruction Set Computing):



(architektura R/M)

A) Czas - w osi poziomej
B) Dane na 3 magistralach: A - magistrala adresowa, B - magistrala danych, C - magistrala sterująca
C) Pionowe linie - *"można traktować jako"* aktywne zbocza sygnału (w zasadzie wszystkie procesory konstruuje się jako automaty synchroniczne)

I. Cykl maszyny - czas potrzebny na wykonanie jednej operacji (jak mamy jedną magistralę musimy najpierw odczytać kod w instrukcji, potem odczytać dane)

Wykonanie jednego rozkazu W MASZYNIE Z JEDNĄ MAGISTRALĄ:

A) Etap pobrania kodu operacji (IF: Instruction Fetch)

1. Procesor wytwarza adres rozkazu (AG: Adres Generation) i wystawia na magistralę zawartość rejestru, który nazywa się licznikiem rozkazów (PC: Programme Counter).
2. Po pewnym czasie pamięć odpowiada kodem tego rozkazu, procesor pobiera kod rozkazu

B) **Etap dekodowania (ID: Instruction Decode)**

1. Odczytywanie i interpretacja tego co pobraliśmy, trzeba rozróżnić rozkaz spośród wielu, chodzi o **wybranie właściwego rozkazu**
2. W tym etapie procesor dowiaduje się też **skąd ma wziąć dane**

C) Etap generowania adresu (AG : Address Generation)

1. Wytworzenie adresów, których rozkaz potrzebuje, czyli wartości, na których rozkaz będzie operował
2. Wystawienie adresy na magistralę adresową

(Generowany adres nie jest czymś prostym, oblicza się go w mniej lub bardziej skomplikowany sposób, jeśli natomiast adres jest podany bezpośrednio w kodzie rozkazu trzeba go jakoś wydobyć z tego kodu rozkazu, rozkaz składa się bowiem z adresu i informacji co chcemy zrobić. Dlatego musi być etap do obliczenia go)

PC+ - schowana w międzyczasie operacja zwiększenia licznika rozkazów, żeby móc nieco szybciej wytworzyć adres kolejnego rozkazu jak skończymy bieżący rozkaz

D)Etap odczytywania/pobrania argumentów (DF : Data Fetch)

1. Procesor **odbiera od pamięci dane**, których potrzebuje jako argumentów
   OA - adres pamięci
   OR - z rejestrów

E) Wykonanie rozkazu (EX : Execution)

1. Wykonanie operacji (trwa zauważalnie mniej niż całe przygotowanie do niego czyli poprzednie

etapy)

F) Zapisanie wyniku

1. Procesor zapisuje wynik, żeby się nie zmarnował

Jak to działa na prostym modelu procesora (przypominającym układ cyfrowy):

LEGENDA:

PAMIĘĆ - pamięć zewnętrzna

Dane - magistrala danych

Adres - magistrala adresowa

Sterowanie - magistrala sygnałów sterujących mówiących czy zapisywać czy odczytywać

Reszta (Procesor):

1. **Ścieżka przepływu danych (na szaro)**

A) **ALU** - Jednostka wykonawcza arytmetyczno-logiczna (Aritmetic Logic Unit):

- Sumatory
- Subtraktory
- Układy mnożące i inne

B) **R** - rejestry

C) **MB/R , TMP/R** - dodatkowe rejestry

D) **CR** - dodatkowe flagi (np. nadmiar, przeniesienie)

2.

A) sterowanie - najważniejszy element w procesorze, automat, który steruje wszystkim dookoła tzn. wystawia właściwe sekwencje sygnałów sterujących mówiących kiedy i co zrobić, wybierana jest w nim na podstawie rozkazu odpowiednia ścieżka

B)IR: Instruction Register - część odpowiedzialna za pobieranie kodu rozkazu

C) dekoder - dekoduje pobrany rozkaz

D) PC - licznik rozkazów, wyznacza adres kolejnego rozkazu Uwaga! Zazwyczaj **nie jest to rejestr** tylko **licznik**, który się zwiększa o ileś co cykl maszynowy

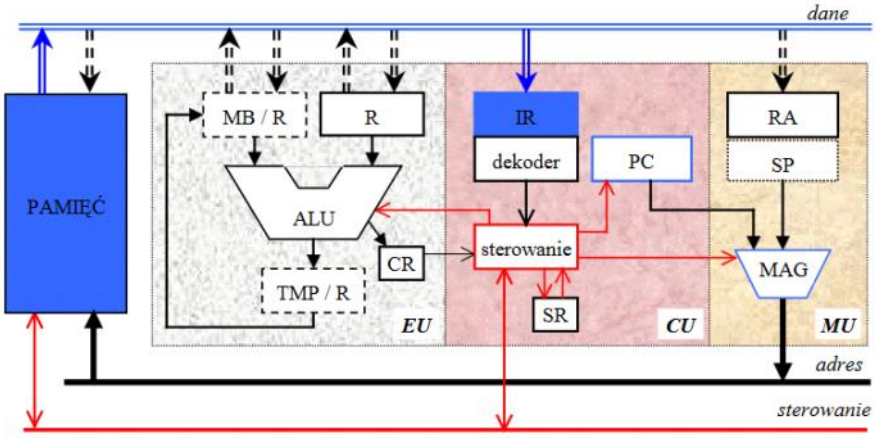
3.

A) RA - rejestr adresowy

B) MAG: Memory Adress Generator - Jednostka wytwarzająca adresy pamieci, osobny układ do obliczania adresów, w zależności od etapu obliczanie odbywa się albo na podstawie PC - licznika rozkazów, albo na podstawie "różnych innych rzeczy w procesorze na podstawie, których obliczane są adresy argumentów"

.....

Etap pobrania kodu operacji (IF: Instruction Fetch):



- *F* (pobranie kodu): adres (PC) → pamięć → rejestr rozkazów IR

Opis:

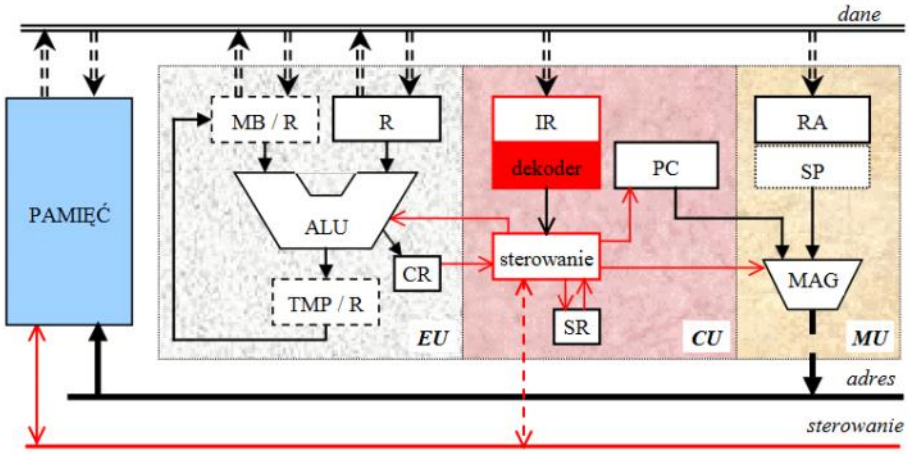
Wystawienie adresu na magistralę adresową obliczanego na podstawie PC

Pamięć odpowiada kodem rozkazu na podstawie magistrali adresowej i sterującej

Kod rozkazu jest zatrzaskiwany w rejestrze rozkazu

.....

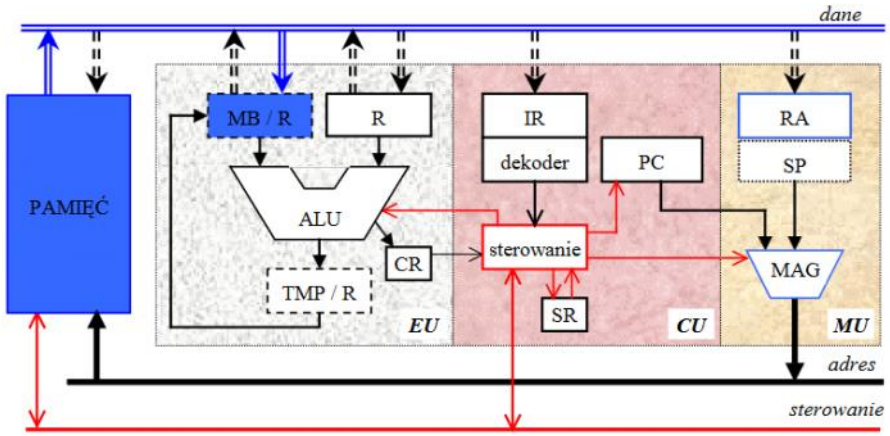
Etap dekodowania (ID: Instruction Decode):



- *D* (dekodowanie) : rejestr rozkazów IR → sterowanie (CR,SR)

Opis:
Rozkaz jest dekodowany

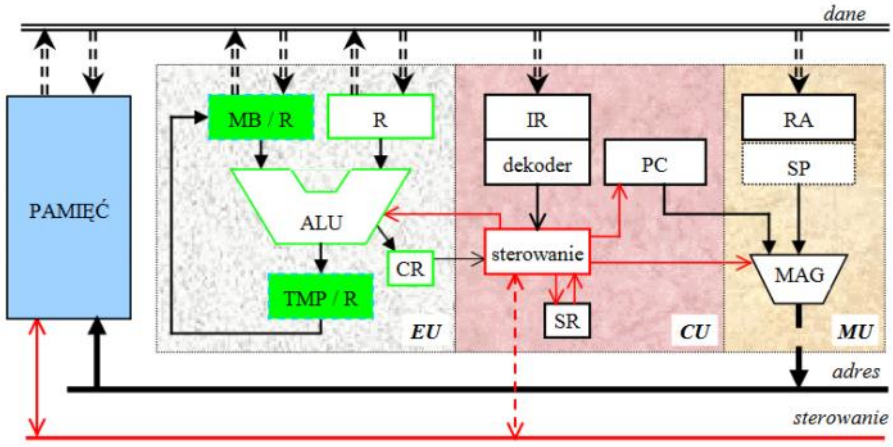
……



- *R* (odczyt danej): adres (RA) → pamięć → bufor (rejestr) MB

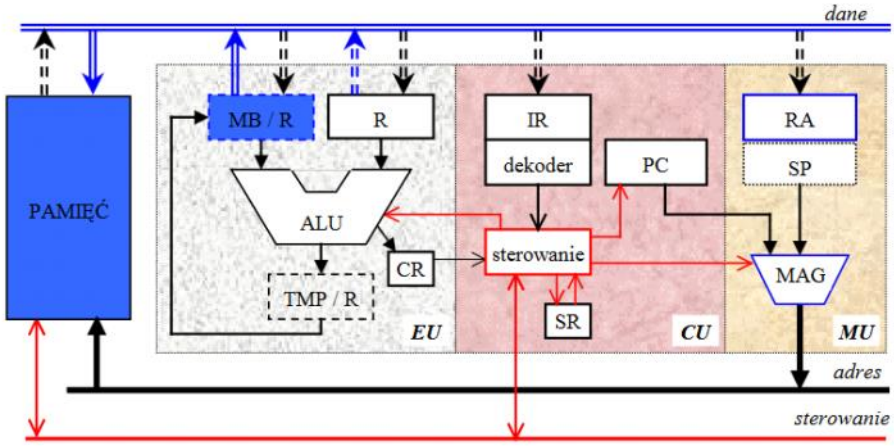
Opis:
Adres jest wytwarzany z innych danych (nie PC), jest wystawiany na magistralę
Pamięć pobiera go i zwraca kod argumentu i zatrzaskuje w dodatkowych rejestrach pomocniczych

……



- *E* (wytworzenie wyniku): ALU (MB/R, R) → bufor TMP/ rejestr R

Opis:
Odczytane dane są przetwarzane i zapisywane do argumentu wynikowego



- *W* (zapis wyniku): bufor MB → pamięć (adres (RA))

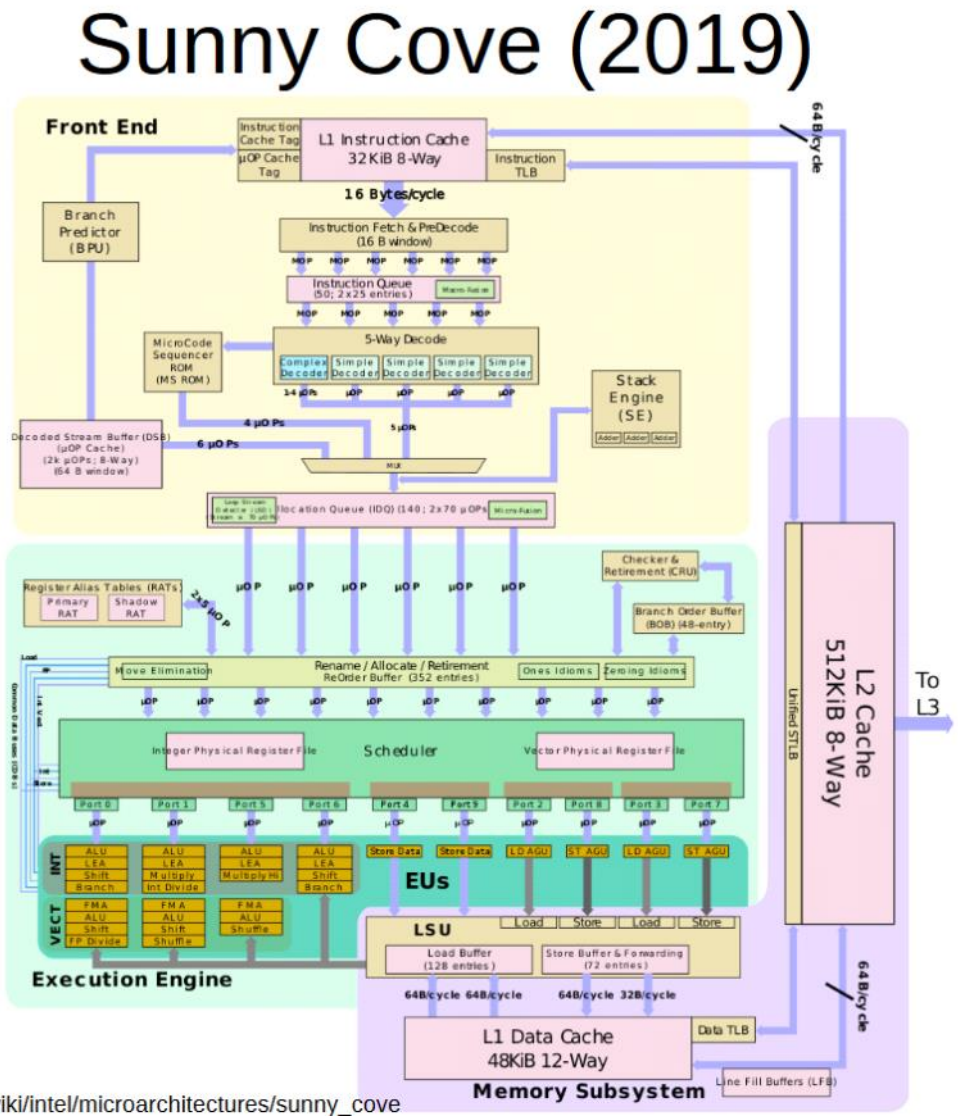
Opis:
Wynik zostaje zapisany do pamięci
Uwagi: Sterowanie działa cały czas.
I tyle z cyklu rozkazowego.

Praktyka

sobota, 18 maja 2024 21:10

W rzeczywistości wysokowydajne maszyny nijak nie pasują do poprzedniego schematu!
(Tak działa jednak mikrokontroler Intel 8051!)

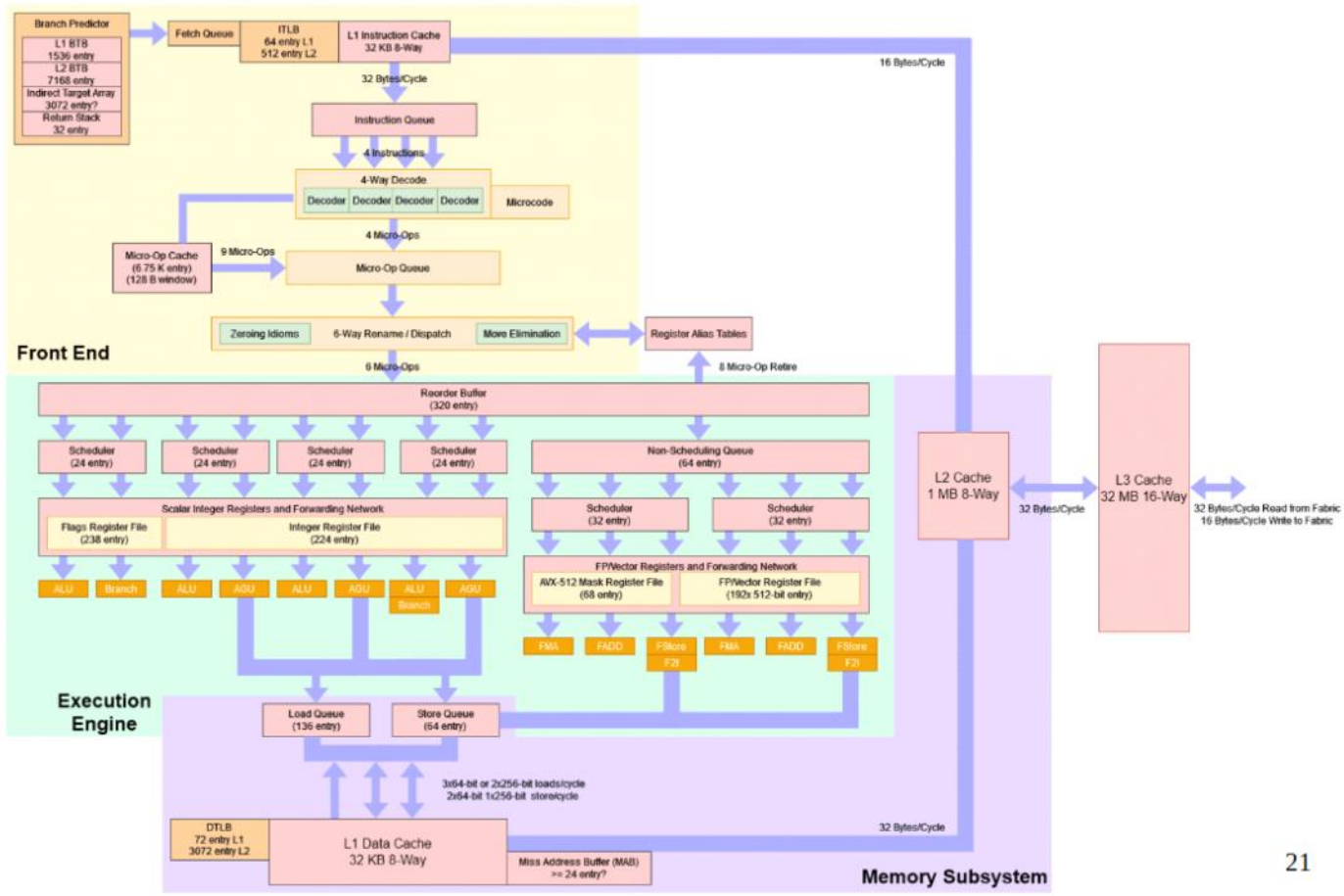
Intel (poprzednia generacja):



/wiki/intel/microarchitectures/sunny_cove
źródło: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

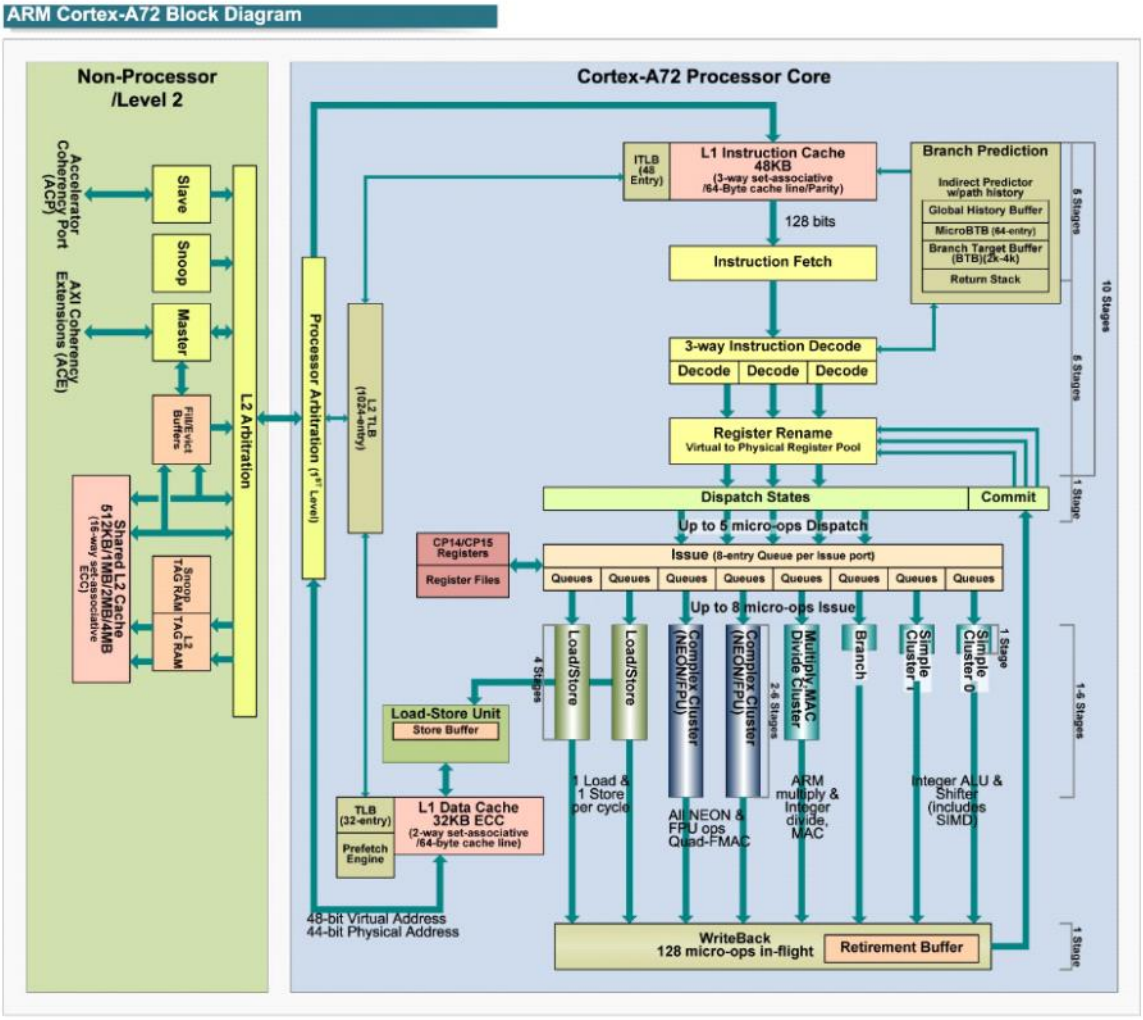
AMD (obecna generacja):

Zen 4 (2022)



źródło: https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/

ARM (sprzed kilku lat):



22

źródło: https://pc.watch.impress.co.jp/docs/column/kaigai/699491.html

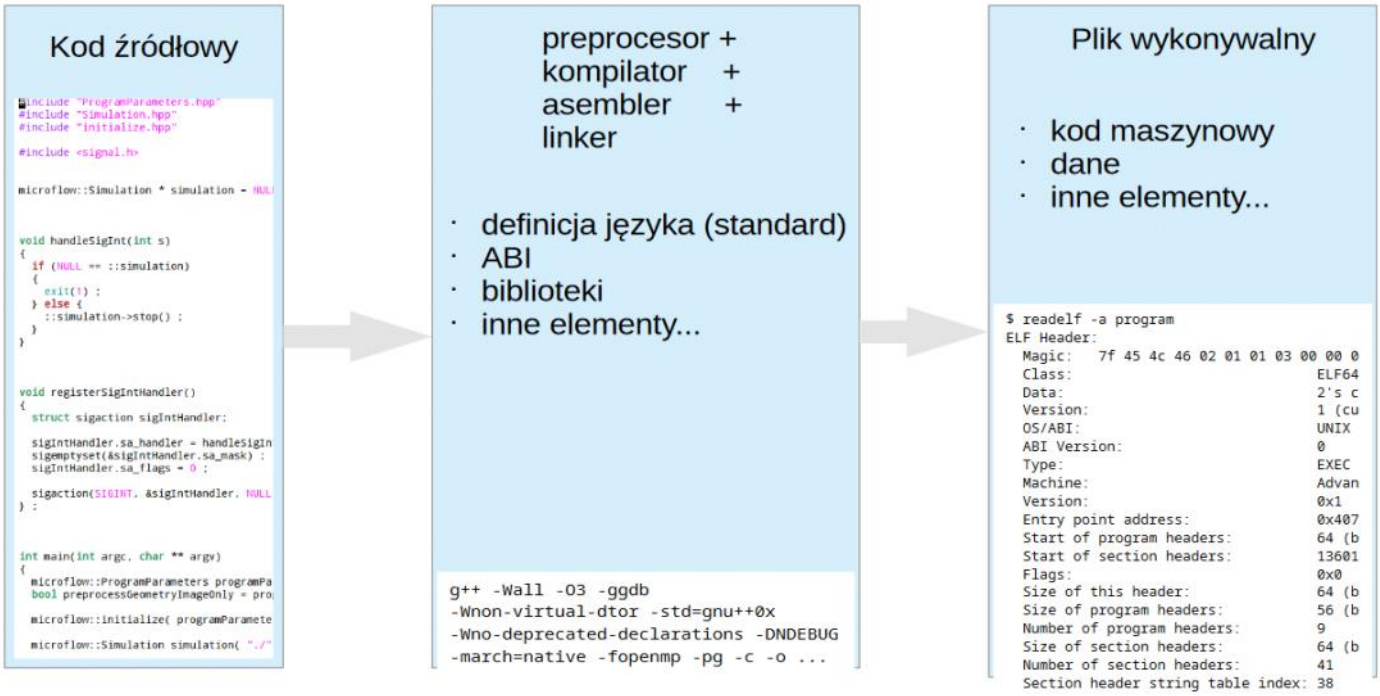
Obecnie procesory składają się z dwóch części:

1. frontend - tłumaczący jeden język na inny język, który można efektywniej wykonać,
2. backend - wykonujący ten inny efektywny język

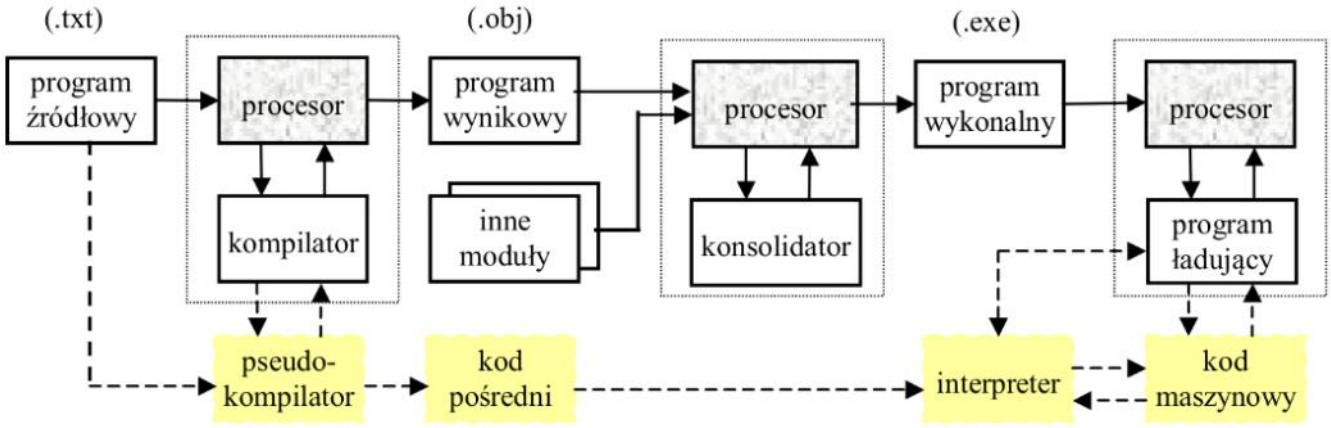
Programowanie

sobota, 18 maja 2024 21:33

Program z naszego punktu widzenia:
Sekwencja jakoś utrwalona, kod źródłowy pisany jako tekst czytelny dla człowieka, kod ten następnie jest przetwarzany do tego co znajduje się na taśmie tzn. w pamięci, odczytywane przez procesor, to jest kompilowane do pliku wykonywalnego, który jest ładowany do pamięci procesora i uruchamiany.



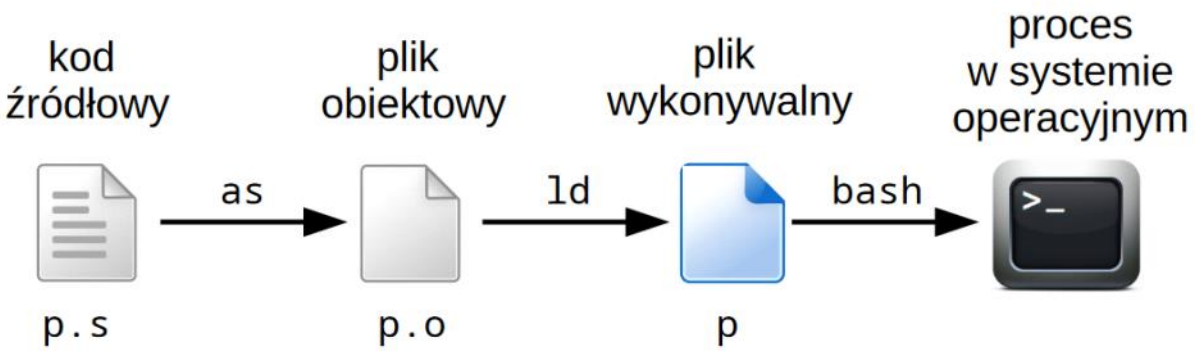
Na komputerze jest uruchamiany specjalny program przetwarzający kod źródłowy na tak zwany plik obiektowy, w językach kompilowanych zazwyczaj realizacja przetwarzania z poziomu kodu źródłowego na plik wykonywalny jest kilkuetapowa z różnych powodów.



My rozróżniamy dwa etapy:

1.Przetworzenie z kodu źródłowego do kodu obiektowego
2.Innym narzędziem np.. Linkerem, konsolidatorem, programem łączącym plik obiektowy lub większa ich ilość są linkowane, łączone, konsolidowane do pliku wykonywalnego i po jego utworzeniu ten plik jest przez system operacyjny ładowany do pamięci, odpowiednio konfigurowany i uruchamiany, na każdym etapie jest wykorzystywany procesor. To jednak dzieje się przed etapem samego działania, który nas najbardziej interesuje.

W językach intepretowanych natomiast kod źródłowy może być przetwarzany do czegoś co można trochę szybciej przetworzyć programowo i z każdym uruchomieniem jest coś w rodzaju maszyny wirtualnej, która odczytuje z pliku kody i to interpretuje czyli wywołuje odpowiednie funkcje po sprawdzeniu co tam było (dwa rzędy wielkości wolniejsze).




```
$ vim p.s
$ as p.s -o p.o && ld p.o -o p && ./p > wy && hexdump -C wy
```

Więcej przykładów w pliku sesja.txt i w materiałach do laboratorium

Opowiada o labach, o nazywaniu plik, o używaniu składni GNU asemblera, który tłumaczy swój plik na plik obiektowy, o linkerze ld przetwarzającym kod obiektowy w plik wykonywalny i następnie za pomocą powłoki systemowej w okienku terminala ten plik będzie ładowany i uruchamiany.

Przykład wyżej w bashu
$ - znak zachęty powłoki systemowej
vim p.s - uruchomienie edytora tekstowego vim, Tomczak tłumaczy, że używa vima, bo często pracuje na zdalnych maszynach, na klastrach komputerów i nie da się szybko zrobić interfejsu graficznego na słabym łączu, chociaż podobno są lepsze, ale nie chce mu się uczyć

$ as p.s -o p.o
Asemblacja, pierwszy argument czyli plik źródłowy oraz drugi oznaczony opcją -o plik wynikowy

&& - operator basha, w skrócie uruchamia to co jest po nim jeśli to co jest wcześniej się uda

ld p.o -o p - uruchomienie linkera ld w podobny sposób

&&

./p uruchamianie programu

> wy - wynik tego programu zostanie zapisane do takiego pliku

&&

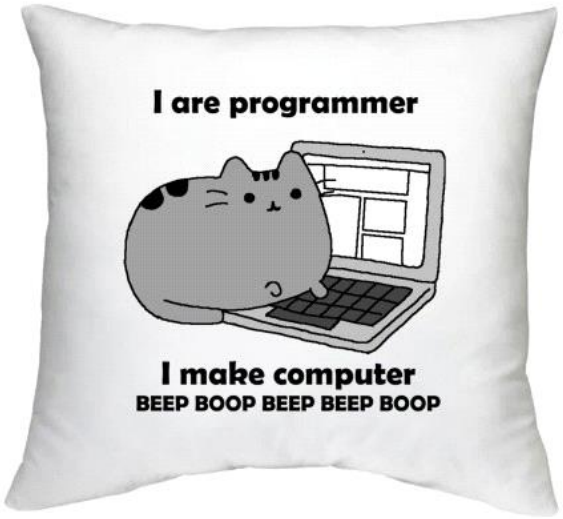
hexdump -C wy - Popatrzymy sobie co w tym pliku jest specjalnym narzędziem

Przykład:

```
$ as -o rw.o rw.s
$ ls -a
.  ..  rw.o  rw.s
$ file rw.o
rw.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not
stripped
$ ld -o rw rw.o
$ ls -a
.  ..  rw  rw.o  rw.s
$ file rw
rw: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, not stripped
$ ./rw
Hello, world!
$ objdump -f rw
rw:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400086
```

Jak może wyglądać ko"t"d?

sobota, 18 maja 2024 22:01




```
$ as -alh rw.s
GAS LISTING rw.s  page 1
   1              # Numbers of kernel functions.
   2              EXIT_NR  = 1
   3              READ_NR  = 3
   4              WRITE_NR = 4
   5
   6              STDOUT = 1
   7              EXIT_CODE_SUCCESS = 0
   8
   9
  10              .text
  11 0000 48656C6C  msg: .ascii "Hello, world!\n"
  11      6F2C2077
  11      6F726C64
  11      210A
  12              msgLen = . - msg
  13
  14
  15              .global _start
  16
  17              _start:
  18
  19 000e B8040000  mov $WRITE_NR, %eax
  19      00
  20 0013 BB010000  mov $STDOUT  , %ebx
  20      00
  21 0018 B9000000  mov $msg     , %ecx
  21      00
  22 001d BA0E0000  mov $msgLen  , %edx
  22      00
  23 0022 CD80      int $0x80
  24
  25
  26 0024 B8010000  mov $EXIT_NR          , %eax
  26      00
  27 0029 BB000000  mov $EXIT_CODE_SUCCESS, %ebx
  27      00
  28 002e CD80      int $0x80
```

31

Pogrubione znaki to to co znajduje się w pamięci procesora , ciągi bajtów.
To na niebiesko to adresy komórek pamięci względem pierwszej komórki.
To w pierwszej kolumnie to numery kolumn kodu źródłowego.

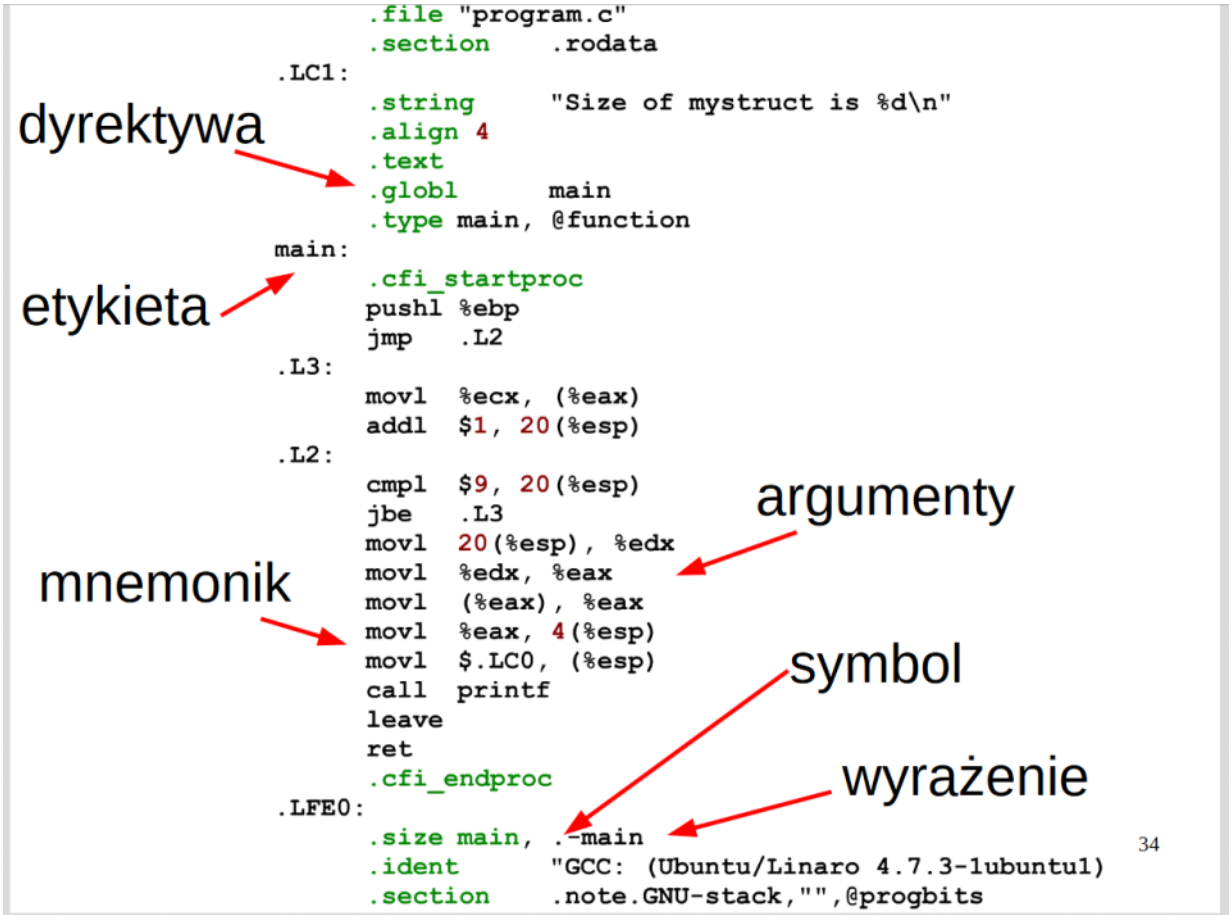
Jak widać niektóre linie kodu źródłowego nie powodują umieszczanie niczego w pamięci, a inne powodują.

Co to jest asembler?

sobota, 18 maja 2024 22:11

- **asembler** – program tłumaczący kod w języku asemblera na kod maszynowy (przykłady dla architektury x86: GNU Assembler *gas*, The Netwide Assembler *nasm*, Turbo Assembler *tasm*, Microsoft Macro Assembler *masm)*
- **język asemblera** – potocznie *asembler*, w dużym uproszczeniu mnemoniczny zapis kodu maszynowego

Tomczak zwraca uwagę na polską pisownię "asembler"

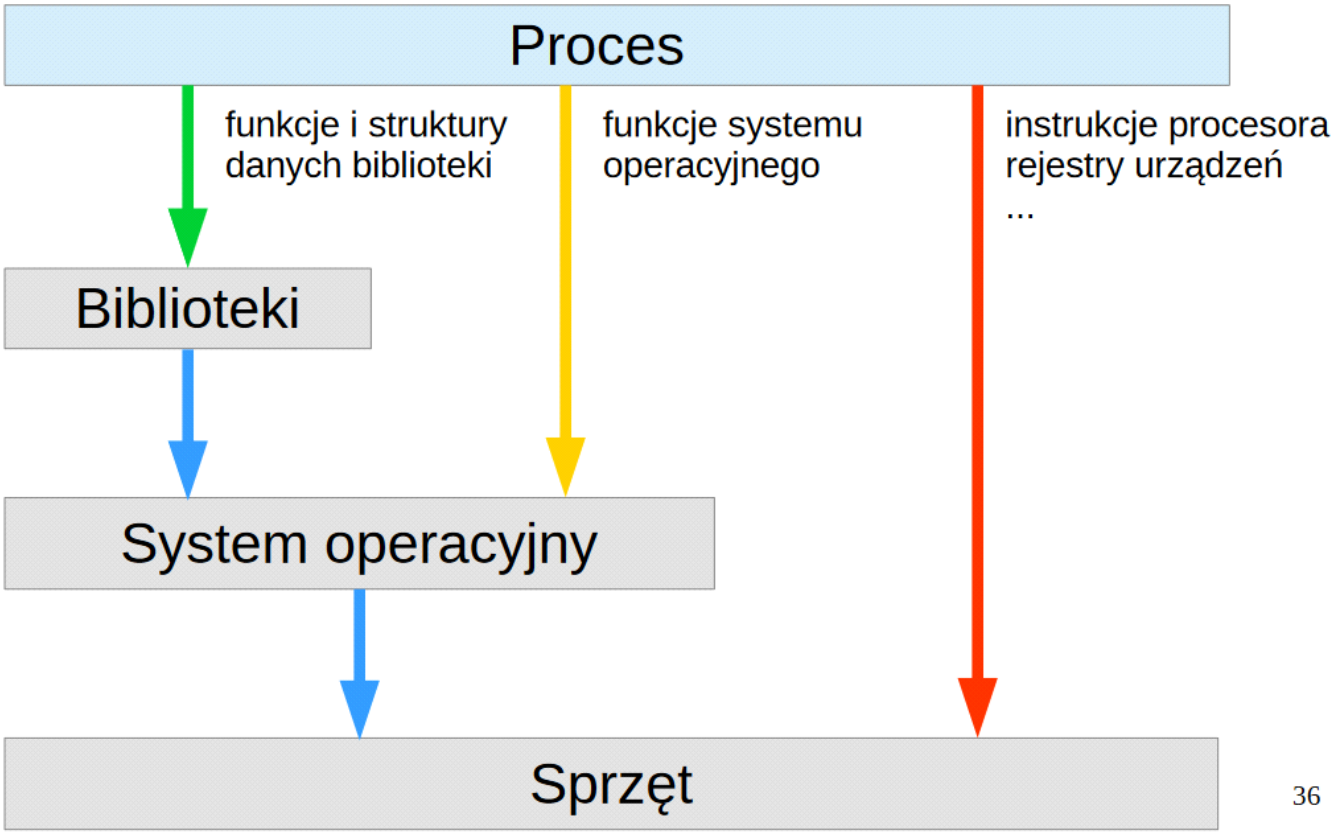


Dyrektywa (z kropką z przodu) - sterują w pewien sposób procesem kompilacji
.globl - jak wynik obiektowy się wyprodukuje to symbol o nazwie main powinien być widoczny dla narzędzi, które będą tego pliku obiektowego używały

Mnemoniki - zrozumiałe częściowo dla człowieka nazwy operacji, które procesor potrafi wykonać

Najważniejsze są argumenty i tryby adresowania!

To co napiszemy w postaci kodu źródłowego i skompilujemy nazywamy **plikiem programu**. Następnie po jego załadowaniu do pamięci i uruchomieniu, a nawet trochę wcześniej zacznie się nazywać **procesem**. Z jednego programu możemy uruchomić wiele procesów czyli działających programów.



36

Kiedy przygotowujemy kod źródłowy, który ma działać jako proces będziemy komunikować się z trzema elementami:

1. Ze sprzętem bezpośrednio (mnemoniki "input", funkcje systemu operacyjnego, z bibliotek kogoś innego z innymi przydatnymi rzeczami) (pominięte)

   Przykład funkcji WRITE oraz sprawdzenia jej opisu na lunuxie:

Interfejsy wysokopoziomowe

```
$ man 2 read
```

```
WRITE(2)                          Linux Programmer's Manual                          WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the file referred to by the file
       descriptor fd.

       The number of bytes written may be less than count if, for example, there is insufficient  space  on
       the  underlying  physical  medium,  or  the  RLIMIT_FSIZE  resource  limit is encountered (see setr-
```

/usr/include/asm/unistd_32.h

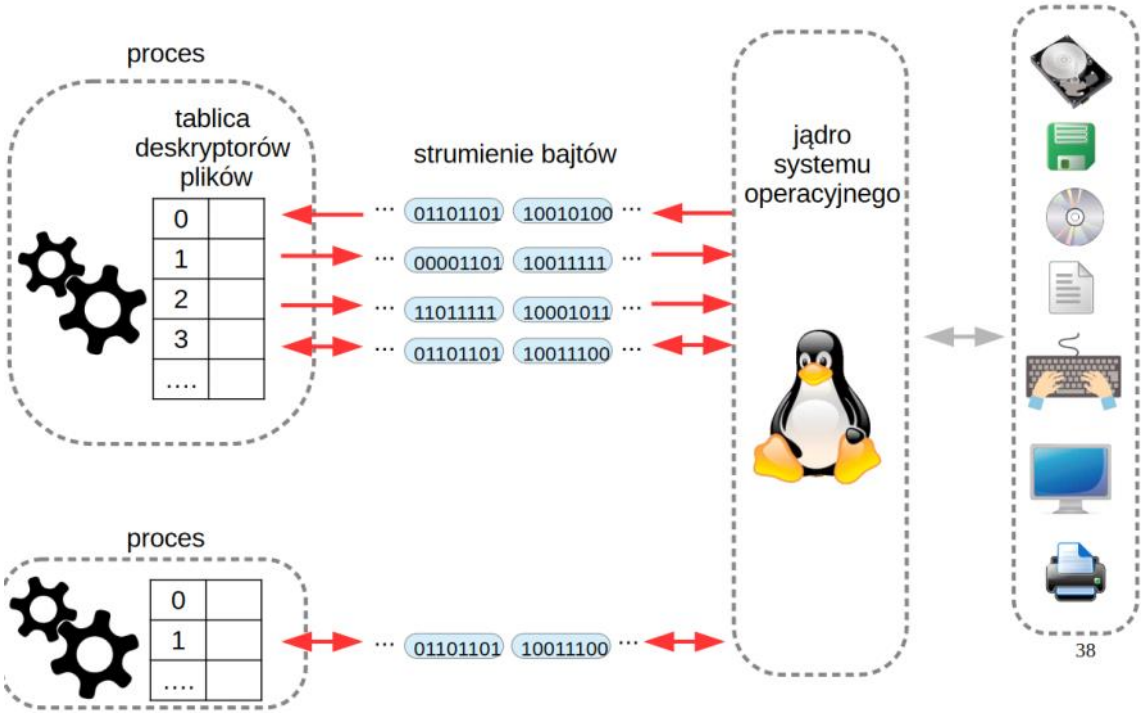
```
1 #ifndef _ASM_X86_UNISTD_32_H
2 #define _ASM_X86_UNISTD_32_H 1
3
4 #define __NR_restart_syscall 0
5 #define __NR_exit 1
6 #define __NR_fork 2
7 #define __NR_read 3
8 #define __NR_write 4
9 #define __NR_open 5
10 #define __NR_close 6
11 #define __NR_waitpid 7
12 #define __NR_creat 8
13 #define __NR_link 9
14 #define __NR_unlink 10
15 #define __NR_execve 11
```

```
24 mov $WRITE_NR, %eax
25 mov $STDOUT  , %ebx
26 mov $msg     , %ecx
27 mov $msgLen  , %edx
28 int $0x80
```

My się jednak nie będziemy na tym skupiać, skupimy się na procesorze i pamięci.

"Jak pingwin w wodzie "

sobota, 18 maja 2024 22:32

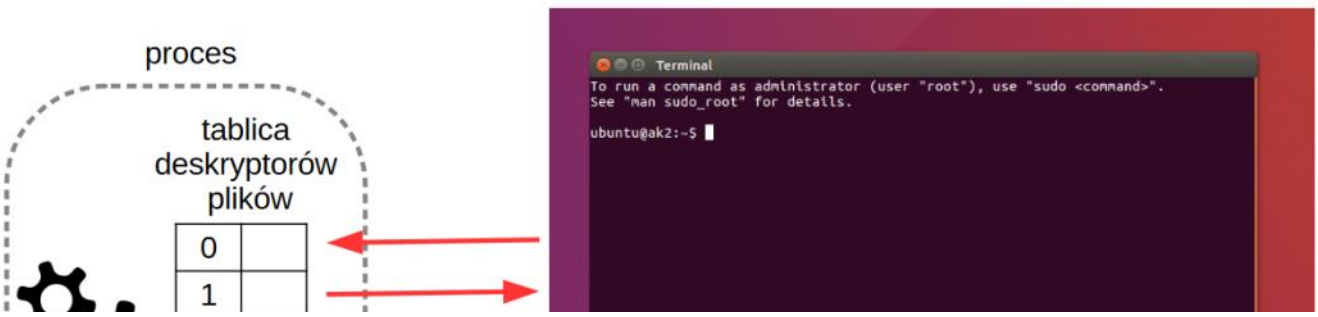


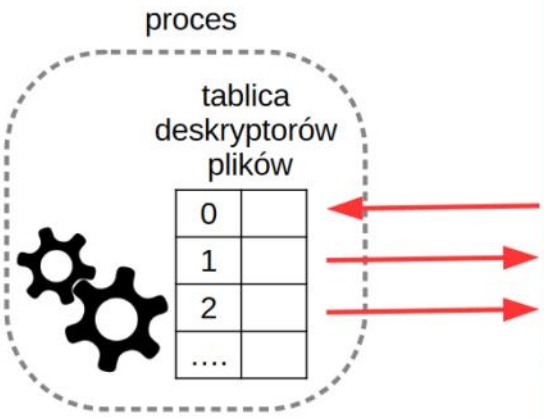
STRUMIEŃ BAJTÓW

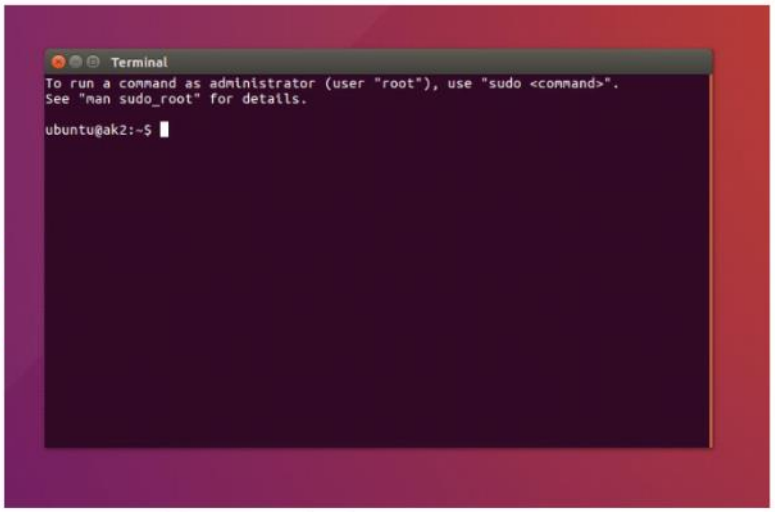
Z każdym urządzeniem i w zasadzie ze wszystkim wiązana jest abstrakcja strumienia bajtów czyli jakie by to urządzenie nie było może być widziane przez proces jako źródło napływających bajtów albo miejsce gdzie można własne bajty wysyłać.

Jak tego się używa?

1. Stworzenie strumienia - stworzenie deskryptora pliku, z punktu widzenia linuxa i unixa te strumienie są obsługiwane przez funkcje do obsługi na plikach, prościej mówiąc: "wszystko jest dostępne jak plik". Można otworzyć taki plik czyli "zażyczyć sobie dostępu" po nazwie. Otrzymuje się deskryptor wyniku otworzenia. Deskryptor to "mała liczba". Można go używać w funkcjach czytających ileś bajtów lub piszących ileś bajtów ciągiem i można to zamknąć. Tyle wystarczy wiedzieć. Nie musimy otwierać i zamykać nowych strumieni jeśli wystarczą nam 3 podstawowe.
2. *Tablica deskryptorów plików * - zawiera informacje gdzie jest dany strumień i jak ma go obsługiwać system operacyjny (nie interesuje nas specjalnie)*
3. Indeksy w tablicy deskryptorów plików - deskryptory (jak widać na przykładzie wyżej małe liczby, bo strumieni jest niewiele)
4. Za pomocą funkcji systemowych czytania i pisania bajtów(READ, WRITE) następuje komunikacja z systemem operacyjnym, który wykonuje potrzebne operacje na różnych rzeczach, urządzeniach przechowujących dane, na plikach, na terminalach, na drukarkach, nawet na pamięci. Potem system może też odsyłać dane.
5. Funkcje READ i WRITE przepisują dane z pamięci do strumienia lub ze strumienia do pamięci, a resztą zajmuje się system operacyjny
6. Ciekawostka: Komunikacja za pomocą strumienia nie musi być z innym urządzeniem, ale może być np. między procesami



7. 



Podczas tworzenia procesu tworzone są trzy strumienie:
0 – standardowy strumień wejściowy
1 – standardowy strumień wyjściowy
2 – standardowy strumień błędów

Proces emulujący terminal.
Wyświetla glify (potocznie obrazki czcionek) na podstawie odbieranych bajtów traktowanych jak kody ASCII lub kody sterujące. Pozwala na edycję wprowadzanych danych dzięki ich buforowaniu.

Uwaga – glify nie są wyświetlane dla części odbieranych bajtów!

40

8. Strumienie umożliwiają dostęp do klawiatury i tego co będzie się wyświetlało na ekranie.
9. <u>Ważne! Pamiętać, że to są strumienie/ciągi bajtów!</u>
10. Terminal:

- Edytor tekstu, w którym się wpisuje różne rzeczy,
- można edytować, przeglądać historię,
- Podczas naciskania "enter" żądanie jest wysyłane przez jądro systemu "gdzieś"
- nie pisze tekstu tylko rysuje obrazki, kiedy przychodzi bajt odpowiadający obrazkowi litery a to rysuje obraz litery a zgodnie ze czcionką
- Ważne! Jeśli program produkuje złe dane tzn. nie tłumaczalne na obrazek to nie zobaczymy efektów
- Domyślnie strumienie w momencie uruchamiania procesu są wiązane z terminalem związanym z procesem

Powłoka systemowa (*ang. shell*) pozwala na przekierowanie strumieni:

https://www.gnu.org/software/bash/manual/html_node/Redirections.html

```
$ ./p < input_data > output_data
```

przekierowanie standardowego wejścia do pliku input_data, przekierowanie standardowego wyjścia do pliku output_data

```
$ ./p < input_data 2>&1 | hexdump -C
```

CHAT GPT (Tomczak pomija): przekierowanie standardowego wejścia do pliku input_data, a wyjścia błędów do standardowego wyjścia wyświetla zawartość wejścia w formacie heksadecymalnym i ASCII.

Wspomniane zostaje związanie standardowych strumieni 0-2 ze strumieniami znanymi z języka c (stdin, stdout i stderror). Tu jest wersja bezpośrednia, a biblioteki c opakowują to innymi strukturami danych.

TYLE Z WPROWADZENIA! DZIĘKUJĘ BARDZO