

Architektury systemów komputerowych

SKRYPT

Anna Karaś
2018-12-02

Są ludzie i pajtonowcy

Wskaźniki w C

liczba - wartość zmiennej
&liczba - adres zmiennej
wskaźnik - wartość wskaźnika, czyli adres wskazywanej przez niego zmiennej
*wskaźnik - wartość wskazywanej przez wskaźnik zmiennej

```
int zmienna;  
int *wsk;  
wsk = &zmienna;  
*wsk = 6; // zmienna = 6;
```

$(i = *wsk) == (i = j)$

prototyp
`void foo(float x, int *y, int *z);`

wywołanie
`foo(3.14, &i, &b);`

zwracanie wskaźników
`int *max (int *a, int *b) {
 return (*a > *b) ? a : b;
}`

wskaźnik na stałą wartość:
`const int *a;`

stały wskaźnik:
`int* const b;`

wskaźnik na tablicę vs wskaźnik na wskaźnik
`int tab[3][5];
int i, j;
tab[i][j] == *(*(tab + i) + j)
tab[0][0] == *(*(tab + 0) + 0) = **tab`

:t tab = int(*)[5]

Lista nr 0

Podstawy

- 1 bajt - najmniejsza adresowalna jednostka informacji pamięci komputerowej, 1B = 8b
- słowo - to podstawowa porcja przetwarzania informacji w systemach komputerowych, określona przez rozmiar rejestrów procesora
- most significant bit (MSB, najbardziej znaczący, najstarszy bit) -> 101100 <- least significant bit (LSB, najmniej znaczący, najmłodszy bit)

Kolejność bajtów (ang. *endianess*)

- big endian - msb first

0x4A3B2C1D

100 101 102 103
4A 3B 2C 1D

- little endian - lsb first, x86

0x4A3B2C1D

100 101 102 103
1D 2C 3B 4A

Operatory bitowe i logiczne w C

- bitowe: koniunkcja &, alternatywa |, xor ^, negacja ~ - działają na poszczególnych bitach
- logiczne: &&, ||, ! - biorą pod uwagę wartość logiczną argumentów
- && i || nie ewaluują drugiego operandu, jeśli pierwszy oblicza się do false
- & i | ewaluują oba operandy

Shifty

- left shift x << y
odrzuca nadmiarowe bity z lewej
uzupełnia zerami z prawej
przesunięcie logiczne
- right shift x >> y
odrzuca nadmiarowe bity z prawej
przesunięcie logiczne - uzupełnia zerami z lewej -
unsigned
przesunięcie arytmetyczne - powiela msb z lewej -
signed
- undefined behavior
shift amount < 0 lub >= wielkość słowa

argument x	0110 0010	argument x	1010 0010
<< 3	0010 0000	<< 3	0001 0000
log. >> 2	0001 1000	log. >> 2	0010 1000
aryt. >> 2	0001 1000	aryt. >> 2	1110 1000

Podstawowe bitwise operations

- k-ty bit zmiennej x
 $(x \gg k) \& 1$
- wyzerowanie k-tego bitu zmiennej x
 $x = x \& \sim(1 \ll k)$
- ustalenie k-tego bitu zmiennej x
 $x = x \mid (1 \ll k)$
- zanegowanie k-tego bitu zmiennej x
 $x = x \wedge (1 \ll k)$
- floor($x / 2^y$)
 $x = (x \gg y)$

Zadanie 1 (6). Przetłumacz poniższą funkcję na procedurę języka C o sygnaturze int num(unsigned).

$$\text{num}(n) = \begin{cases} i & \text{dla } n = 2 \cdot i \\ -i & \text{dla } n = 2 \cdot i + 1 \end{cases}$$

```
int num(unsigned n) {  
    return ((n >> 1) ^ (- (n & 1))) + (n & 1);  
}
```

Zadanie 2 (6). Przetłumacz poniższą funkcję na procedurę języka C o sygnaturze int avg(int, int).

$$\text{avg}(x, y) = \left\lfloor \frac{x + y}{2} \right\rfloor$$

```
int avg(int x, int y) {  
    return (x >> 1) + (y >> 1) + (x & y & 1);  
}
```

Zadanie 1 (8). Przetłumacz poniższą funkcję na procedurę języka C.

$$\text{fit}(x, y) = \begin{cases} 1 & \text{dla } x + y > \text{INT_MAX} \\ -1 & \text{dla } x + y < \text{INT_MIN} \\ 0 & \text{w p.p.} \end{cases}$$

```
int fit(int x, int y) {  
    int s = (unsigned)x + (unsigned)y;  
    unsigned o = s & ~ (x | y); /* przepelnienie */  
    unsigned u = ~s & x & y; /* niedomiar */  
    return (o >> 31) - (u >> 31);  
}
```

Zadanie 2 (6). Zaprogramuj w języku C procedurę zliczającą zapalone jedynki w każdym z bajtów podanego słowa. Użyj techniki *dziel i zwyciężaj*. Przykład: `popcnt8(0xC00010FF) == 0x02000108`.

```
unsigned popcnt8(unsigned x) {
    unsigned m1 = 0x55555555;
    unsigned m2 = 0x33333333;
    unsigned m3 = 0xFFFF FFFF;
    x = (x & m1) + ((x >> 1) & m1);
    x = (x & m2) + ((x >> 2) & m2);
    return (x & m3) + ((x >> 4) & m3);
}
```

Zadanie 1 (10). Przetłumacz poniższą funkcję na procedurę języka C.

$$\text{adds}(x, y) = \begin{cases} \text{INT_MAX} & \text{dla } x + y > \text{INT_MAX} \\ \text{INT_MIN} & \text{dla } x + y < \text{INT_MIN} \\ x + y & \text{w p.p.} \end{cases}$$

Wskazówka: Rozwiążanie wzorcowe używa 12 operatorów, nie licząc przypisania. Można użyć porównania, ale nie na pełną liczbę punktów.

```
1 int adds(int x, int y) {
2     int sum = x + y;
3     int over = ((~(x ^ y)) & (x ^ sum)) >> 31;
4     return (over & ((sum >> 31) ^ INT_MIN)) | (~over & sum);
5 }
```

Zadanie 2 (7). Zaprogramuj w języku C procedurę sprawdzającą, czy słowo jest palindromem bitowym. Jej wynikiem jest wartość logiczna, która oznacza czy dla $i = 0 \dots 15$ zachodzi $x_i = x_{31-i}$.

Wskazówka: Rozwiążanie wzorcowe używa 24 operatorów, nie licząc przypisania.

```
1 unsigned palindrome(unsigned x) {
2     unsigned hi = x >> 16;
3     unsigned lo = x & 0xFFFF;
4     hi = ((hi & 0xFF00) >> 8) | ((hi & 0x00FF) << 8);
5     hi = ((hi & 0xF0F0) >> 4) | ((hi & 0x0F0F) << 4);
6     hi = ((hi & 0xCCCC) >> 2) | ((hi & 0x3333) << 2);
7     hi = ((hi & 0xAAAA) >> 1) | ((hi & 0x5555) << 1);
8     return !(lo ^ hi);
9 }
```

Zadanie 1 (10). Dla x i y typu `unsigned` określamy funkcję `doz` (ang. *difference or zero*) następująco:

$$\text{doz}(x, y) = \begin{cases} x - y, & \text{gdy } x \geq y; \\ 0, & \text{w p.p.} \end{cases}$$

W puste pola wpisz odpowiednie wyrażenia tak, by:

$$\text{mask} = \begin{cases} 0, & \text{gdy } x \geq y; \\ \text{UINT_MAX}, & \text{w p.p.} \end{cases}$$

i by wartość zwracana przez funkcję `doz` była zgodna z powyższą specyfikacją. W wyrażeniach poza stałymi i podanymi zmiennymi możesz użyć wyłącznie operatorów bitowych oraz rzutowania na typ `signed`.

```
unsigned doz(unsigned x, unsigned y) {
    unsigned diff = x - y;
    unsigned mask = (signed)((~x & y) | ((~x ^ y) & diff)) >> 31;
    return diff & ~mask;
}
```

Zadanie 3 (10). Dla ciągu bitów $\vec{x} = (x_{31}, \dots, x_0)$ określamy funkcję $\text{tzm}(\vec{x}) = \vec{y}$ (ang. *trailing zeros mask*), której wynikiem jest taki ciąg bitów $\vec{y} = (y_{31}, \dots, y_0)$, że dla $i = 0 \dots 31$ zachodzi:

$$y_i = \begin{cases} 0, & \text{jeśli } x_j = 0 \text{ dla wszystkich } j = 0, 1, \dots, i; \\ 1, & \text{w p.p.} \end{cases}$$

Przykład: `tzm(0x68fa4560) = 0xfffffffffe0.`

W prostokąt poniżej wpisz ciąg co najwyżej pięciu instrukcji wyrażeniowych tak, by powstała poprawna definicja funkcji `tzm`. W wyrażeniach poza zmienną `x` i stałymi możesz użyć wyłącznie operatorów bitowych i operatora przypisania, przy czym łączna liczba wystąpień operatorów nie może przekraczać 15.

```
unsigned tzm(unsigned x) {
```

```
    x |= x << 1;
    x |= x << 2;
    x |= x << 4;
    x |= x << 8;
    x |= x << 16;
```

```
}
```

W prostokąt poniżej wpisz wyrażenie zawierające wyłącznie zmienne `x` i `m`, stałe oraz operatory bitowe tak, by powstała poprawna definicja funkcji, która zwiększa swój argument o 1:

```
unsigned increment(unsigned x) {
```

```
    unsigned m = tzm(~x);
```

```
    return (x & m) | (m ^ (m << 1));
}
```

Reprezentacja programów

Kod trójkowy (ang. *three-address code*, TAC)

- to postać pośrednia stosowana przez kompilatory przy translacji z języka wysokiego poziomu do asemblera
- w TAC nie ma wysokopoziomowych instrukcji sterujących (for, while, switch), typów złożonych, procedur

Kod w TAC składa się z

- adresów:
 - stałe
 - nazwy (zmiennej, funkcji, etykiety)
 - zmienne tymczasowe
- instrukcji:

```
- x := y binop z
- x := unop z
- x := y
- goto L
- if b goto L
- if x relop y goto L // skok do L, jeśli x jest w relacji relop do y
- x := &y //wyznaczenie wskaźnika do zmiennej (referencja)
- x := *y, *x := y //dereferencja wskaźnika
- param x //użyj x jako parametru procedury
- call p, n //wołanie procedury p z n argumentami
- return n
```

- wskaźniki w TAC

```
(x := a[i]) == (t := a + i; x := *t)
(a[i] := x) == (t := a + i; *t := x)
```

Typy w TAC

- arytmetyka na wskaźnikach TAC jest beztypowa; a[i] oznacza dostęp do adresu a + i, a nie do i-tego elementu tablicy
- zachowujemy typ wskaźnika, by odwołać się do słowa określonego rozmiaru

```
uint32_t *a;
...
int i = 0;
while (a[i] < v) {
    i++;
}
```

```
L:    i := 0
      t1 := i * 4
      t2 := a[t1]
      if t2 >= v goto E
      i := i + 1
      goto L
E:
```

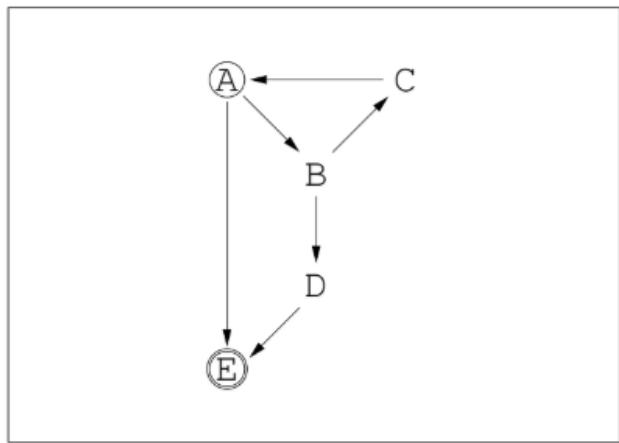
Graf kontroli przepływu

- graf skierowany reprezentujący wszystkie ścieżki programu, które można przejść w trakcie jego wykonania
- wierzchołkami są bloki podstawowe, czyli sekwencje instrukcji za wyjątkiem skoków, kończące się instrukcją skoku
- instrukcje w bloku podstawowym zawsze zaczynamy wykonywać od pierwszej

Zadanie 4 (10). Przetłumacz poniższą procedurę do kodu trójadresowego (TAC), który należy umieścić w większej kratce, po czym oznacz na nim bloki podstawowe. W mniejszą kratkę wrysuj graf przepływu sterowania.

```
1 typedef struct node {
2     long key;
3     struct node *next;
4 } node_t;
5
6 void insert(node_t **next_p, node_t *node) {
7     for (; *next_p; next_p = &(*next_p)->next) {
8         if ((*next_p)->key > node->key) {
9             node->next = (*next_p)->next;
10            break;
11        }
12    }
13    *next_p = node;
14 }
```

UWAGA: Oznacz węzły, w których procedura zaczyna (kółkiem) albo kończy swe wykonanie (podwójnym kółkiem).



UWAGA: W TAC występują wyłącznie wskaźniki na typy maszynowe. Przy operacjach arytmetycznych wszystkie wskaźniki zachowują się jak «`char **`». Typ wskaźnika ma znaczenie tylko przy dereferencji.

```
void insert(node_t **next_p, node_t *node) {
    node_t *t1, *t8;
    node_t **t6, **t7;
    long *t2, *t3;
    long t4, t5

A:   t1 := *next_p;
      if t1 == 0 goto E;

B:   t2 := t1 + 0;          /* &(*next_p)->key */
      t3 := node + 0;        /* &node->key */
      t4 := *t2;
      t5 := *t3;
      if t4 > t5 goto D;

C:   next_p := t1 + 8;     /* &(*next_p)->next */
      goto A;

D:   t6 := t1 + 8;          /* &(*next_p)->next */
      t7 := node + 8;        /* &node->next */
      t8 := *t6;
      *t7 := t8;
      goto E;                /* opcjonalnie */

E:   *next_p := node;
      return;
}
```

Lista nr 1

Data structure alignment

- data alignment - układanie danych w pamięci na adresach podzielnych przez daną liczbę
- data structure padding - dodatkowe bajty danych wstawiane między końcem ostatniego elementu a początkiem nowego, by nowy zaczynał się na adresie wyrównanym do odpowiedniej liczby

Padding jest wstawiany:

- za elementem, którego wyrównanie jest mniejsze od wyrównania elementu następującego po nim
- za ostatnim elementem w strukturze, by rozmiar całej struktury był liczbą podzielną przez największy alignment spośród elementów struktury

Adres struktury jest adresem pierwszego elementu. Nie ma wiodącego paddingu.

```
struct MixedData {  
    char d1;  
    short d2;  
    int d3;  
    char d4;  
}  
8 bytes before compilation
```

```
struct AfterCompilation {  
    char d1;           //1B  
    char padding1[1]; //1B  
    short d2;          //2B  
    int d3;            //4B  
    char d4;           //1B  
    char padding2[3]; //3B  
}  
12 bytes after compilation
```

```
struct FinalPadShort {  
    short s;  
    char n[3];  
};  
size of the structure == 6 (alignment(short) == 2)
```

- nested structs - jeśli struktura ma w sobie zagnieżdżoną strukturę, wewnętrzna również musi być wyrównana do największego alignmentu w strukturze

```
struct foo5 {  
    char c;  
    struct foo5_inner {  
        char *p; //8B  
        short x; //2B  
    } inner;  
};  
  
struct foo5 {  
    char c;           //1B  
    char pad1[7];   //7B  
    struct foo5_inner {  
        char *p;     //8B  
        short x;    //2B  
        char pad2[6]; //6B  
    } inner;  
};
```

- unia - rozmiar unii jest wielkości jej największego elementu (unia pełni rolę monady Either w Haskellu lub typu option w Ocamlu)

Zadanie 3 (7). Posługując się ABI dla architektury x86-64 wyznacz rozmiar struktury node, rozmiary pól i ich przesunięcie względem początku struktury. W puste kratki wpisz odpowiednio przesunięcie i rozmiar danego pola. Zminimalizuj rozmiar struktury biorąc pod uwagę, że bieżący wariant wewnętrznej unii jest wybierany na podstawie pola «flags». W prostokąt po prawej stronie wpisz definicję nowej struktury wraz z przesunięciami i rozmiarami pól.

0	8
8	1
16	8
24	4
24	4
32	8
40	2

```

struct node {
    struct node *next;
    unsigned char flags;
    int (*callback)(struct node *);
    union {
        int errno;
        struct {
            int length;
            char *data;
        };
    };
    short key;
};

/* sizeof(struct node) == 48 */

```

```

struct node1 {
    /* 0 8 */ struct node1 *next;
    /* 8 8 */ int (*callback)(struct node1 *);
    /* 16 8 */ char *data;
    union {
        /* 24 4 */ int errno;
        /* 24 4 */ int length;
    };
    /* 28 2 */ short key;
    /* 30 1 */ unsigned char flags;
    /* 31 1 */ padding */
};

/* sizeof(struct node1) == 32 */

```

Zadanie 2 (8). W tym zadaniu korzystamy z ABI dla architektury x86-64. W puste kratki po lewej stronie wpisz, odpowiednio, rozmiary pól i ich przesunięcia względem początku struktury packet oraz rozmiar całej struktury. Przyjmij przy tym, że stała N ma wartość 7. Rozmiary i przesunięcia wyraź w bajtach. W prostokąt po prawej stronie wpisz wersję struktury o takim samym rozmiarze, jak wyliczony dla oryginalnej struktury packet, zoptymalizowaną tak, by wartość stałej N (rozmiar bufora s_buffer) była największa z możliwych.

0	2
8	8
16	1
17	7
16	4
24	8

```

struct packet {
    short flags;
    struct packet *next;
    union {
        struct {
            unsigned char s_size;
            char s_buffer[N];
        };
        struct {
            unsigned l_size;
            char *l_buffer;
        };
    };
};

/* sizeof(struct packet) == 32 */

```

```

struct packet_opt {
    struct packet_opt *next; /* 0 8 */
    union {
        short flags; /* 8 2 */
        struct {
            short s_flags; /* 8 2 */
            unsigned char s_size; /* 10 1 */
            char s_buffer[21]; /* 11 21 */
        };
        struct {
            short l_flags; /* 8 2 */
            unsigned l_size; /* 12 4 */
            char *l_buffer; /* 16 8 */
        };
    };
};

```

Wskazówka: Zminimalizuj nieużytki powstające wokół pola flags.

volatile

Jakie jest działanie volatile w stosunku do zmiennych?

- zadeklarowanie zmiennej ze słowem kluczowym volatile sprawia, że zawsze zostanie odczytana z pamięci, ponieważ jej wartość może się zmienić pomiędzy dostępnymi (nawet, gdy nie wygląda na zmodyfikowaną)
- powstrzymuje to niektóre optymalizacje mogące bez niego powodować nieprawidłowe działanie programu

Kiedy programiści muszą go użyć, by program zachowywał się poprawnie?

- w praktyce tylko trzy typy zmiennych mogą zmienić swoją wartość niespodziewanie – gdy zmienna jest:
 - rejestrem bądź interfejsem sprzętowym zmapowanym na pamięć
 - zmienną globalną modyfikowaną przez procedurę obsługi przerwania
 - zmienną globalną modyfikowaną przez wiele wątków

static

Jaki jest skutek użycia static w stosunku do zmiennych globalnych, lokalnych i procedur? Kiedy należy go używać?

- zmienne globalne i procedury – w C plik źródłowy odgrywają rolę modułów. Każda zmienna globalna lub funkcja zadeklarowana ze słowem static jest prywatna dla tego modułu. Bez niego jest publiczna i może być używana w dowolnej innej jednostce translacji (pliku źródłowym).
- zmienne lokalne – w danym bloku programu posiadają dokładnie jedną instancję i istnieją przez cały czas działania programu

restrict

Jaką rolę pełni restrict odnośnie typów wskaźnikowych?

- gwarancja, że w czasie istnienia wskaźnika dostęp do wskazywanego przez niego obiektu będzie realizowany wyłącznie za pośrednictwem tego wskaźnika

Reprezentacja danych - liczby całkowite

Example data representations

C data type	typical 32-bit	x86-64
char	1	1
short	2	2
int	4	4
long	4	8
float	4	8
double	8	8
long double	-	10/16
pointer	4	8

Reprezentacja w kodzie uzupełnień do dwóch
http://eduinf.waw.pl/inf/alg/006_bin/0018.php

Konwersja typu między signed a unsigned

- jawna:

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

- niejawna:

```
tx = ux;  
uy = ty;
```

- jeśli wyrażenie zawiera signed i unsigned int, int jest niejawnie konwertowany do unsigned

Constant ₁	Constant ₂	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Rozszerzenie (ang. *expanding*) (np. short int do int)

- unsigned: dodajemy zera
- signed: sign extension

Obcięcie (ang. *truncating*) (np. unsigned do unsigned short)

- unsigned: operacja mod
- signed: podobne do mod

Konwersja między liczbami ze znakiem i bez znaku

$$T2U_w(x) = x_{w-1} \cdot 2^w + x = \begin{cases} x + 2^w & \text{dla } x < 0 \\ x & \text{dla } x \geq 0 \end{cases}$$

$$U2T_w(u) = -u_{w-1} \cdot 2^w + u = \begin{cases} u & \text{dla } u < 2^{w-1} \\ u - 2^w & \text{dla } u \geq 2^w \end{cases}$$

Dzielenie całkowitoliczbe realizowane przez procesor:

$$x \div y = \begin{cases} \lfloor x/y \rfloor & \text{dla } x \cdot y \geq 0 \wedge y \neq 0 \\ \lceil x/y \rceil & \text{dla } x \cdot y < 0 \wedge y \neq 0 \\ \perp & \text{dla } y = 0 \end{cases}$$

$$x \% y = \begin{cases} x - \lfloor x/y \rfloor \cdot y & \text{dla } y > 0 \\ x - \lceil x/y \rceil \cdot y & \text{dla } y < 0 \\ \perp & \text{dla } y = 0 \end{cases}$$

Reprezentacja danych – liczby zmiennopozycyjne

Zadanie 5 (6). Niech i będzie niezerową liczbą całkowitą typu int, oraz f i g niezerowymi liczbami zmiennopozycyjnymi typu float. Podaj dowolne wartości, dla których poniższe wyrażenia będą prawdziwe:

$$i * i < i$$

$$i = INT_MAX$$

$$i \gg 2 != i / 4$$

$$i = -3$$

$$f / 2.0 == f / -2.0$$

$$f = \text{najmniejsza dodatnia zdemormalizowana}$$

$$(f + g) / (f + g) != 1.0$$

$$f = 1.0, g = -1.0$$

Programowanie niskopoziomowe: podstawy

ISA

model programowy procesora (architektura procesora, ang. *instruction set architecture*)

- interfejs pomiędzy software a hardware
- ISA definiuje wszystko co programista języka maszynowego potrzebuje wiedzieć, by napisać program
 - wspierane typy danych
 - what state there is (main memory and registers) and their semantics (addressing modes)
 - instruction set (the set of machine instructions that comprises a computer's machine language)
- historia
 - Intelowe ISA IA32 dominuje rynek przez wiele lat
 - postęp technologii komputerowej wymusza powiększenie wielkości słowa, logicznym następcą architektury 32-bitowej jest 64-bitowa
 - pierwsze podejście Intela to IA64 na procesorach Itanium – niewypał, bo co prawda była wstępnie kompatybilna z IA32, ale był słaby performance
 - AMD wkracza do gry z „x86-64”, bazującym na (intelowym) IA32, ale rozszerzającym go o m. in. nowe typy danych, i oczywiście podwajającym wielkość słowa
 - AMD zmienia nazwę swojej architektury na „AMD64”
 - Intel odnosi porażkę ze swoim IA64, decyduje się na swoją wersję x68-64. Nazywa ją „Intel64”.
- przykłady ISA: ARM, AVR, IA32, Itanium (IA64), MIPS, PowerPC

microarchitecture – implementacja ISA (rozmiary cache, częstotliwość zegara CPU)

ABI

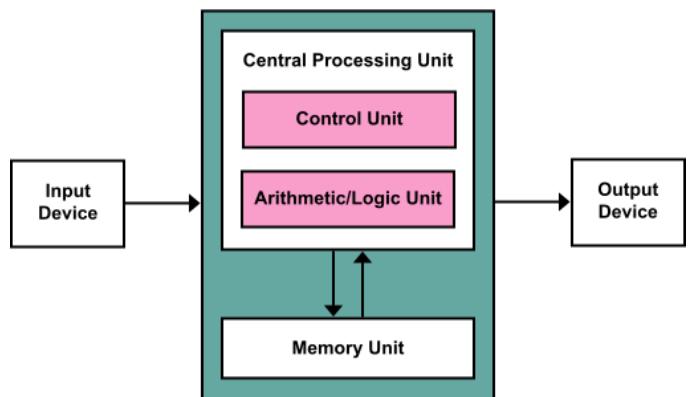
interfejs binarny aplikacji (ang. *application binary interface*)

- interfejs pomiędzy dwoma binarnymi modułami programu; zazwyczaj jeden z nich to biblioteka lub sysopiek, drugi to uruchomiony program usera
- A common aspect of an ABI is the calling convention, which determines how data is provided as input to or read as output from computational routines; examples are the x86 calling conventions.
- ABI covers details such as:
 - a processor instruction set (with details like register file structure, stack organization, memory access types, ...)
 - the sizes, layouts, and alignments of basic data types that the processor can directly access the calling convention, which controls how functions' arguments are passed and return values are retrieved; for example, whether all parameters are passed on the stack or some are passed in registers, which registers are used for which function parameters, and whether the first function parameter passed on the stack is pushed first or last onto the stack
 - how an application should make system calls to the operating system and, if the ABI specifies direct system calls rather than procedure calls to system call stubs, the system call numbers
 - and in the case of a complete operating system ABI, the binary format of object files, program libraries and so on
- ABI vs API:
 - ABI dotyczy oprogramowania w wersji binarnej, a nie w formie kodu źródłowego (API)
 - An ABI defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format
 - API defines this access in source code, which is a relatively high-level, relatively hardware-independent, often human-readable format
- ABI vs ISA:
 - ABI (Application Binary Interface) refers to the calling conventions between functions, meaning what registers are used and what sizes the various C data types are.
 - ISA (Instruction Set Architecture) refers to the instructions and registers a CPU has available.

Architektura von Neumana

architektura von Neumana

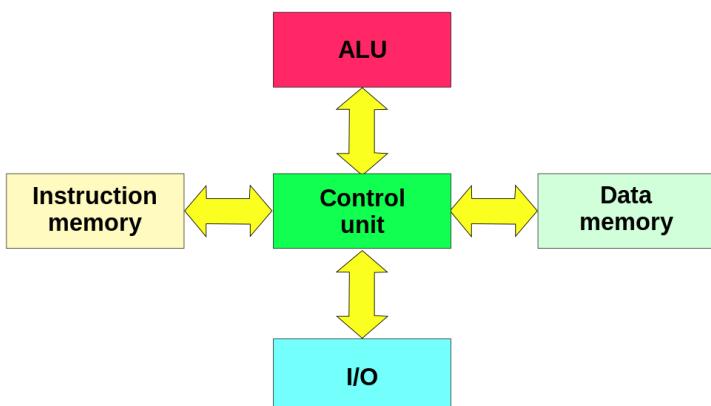
- cechą charakterystyczną tej architektury jest to, że dane przechowywane są wspólnie z instrukcjami, co sprawia, że są kodowane w ten sam sposób
- w architekturze tej komputer składa się z czterech głównych komponentów:
 - pamięci komputerowej przechowującej dane programu oraz instrukcje programu; każda komórka pamięci ma unikatowy identyfikator nazywany jej adresem
 - jednostki sterującej odpowiedzialnej za pobieranie danych i instrukcji z pamięci oraz ich sekwencyjne przetwarzanie
 - jednostki arytmetyczno-logicznej odpowiedzialnej za wykonywanie podstawowych operacji arytmetycznych
 - urządzeń wejścia/wyjścia służących do interakcji z operatorem
- jednostka sterująca wraz z jednostką arytmetyczno-logiczną tworzą procesor
- system komputerowy zbudowany w oparciu o architekturę von Neumanna powinien:
 - mieć skończoną i funkcjonalnie pełną listę rozkazów
 - mieć możliwość wprowadzenia programu do systemu komputerowego poprzez urządzenia zewnętrzne i jego przechowywanie w pamięci w sposób identyczny jak danych
 - dane i instrukcje w takim systemie powinny być jednakowo dostępne dla procesora
 - informacja jest tam przetwarzana dzięki sekwencyjnemu odczytywaniu instrukcji z pamięci komputera i wykonywaniu tych instrukcji w procesorze
- użycie:
 - desktop computers
 - laptops
 - high performance computers



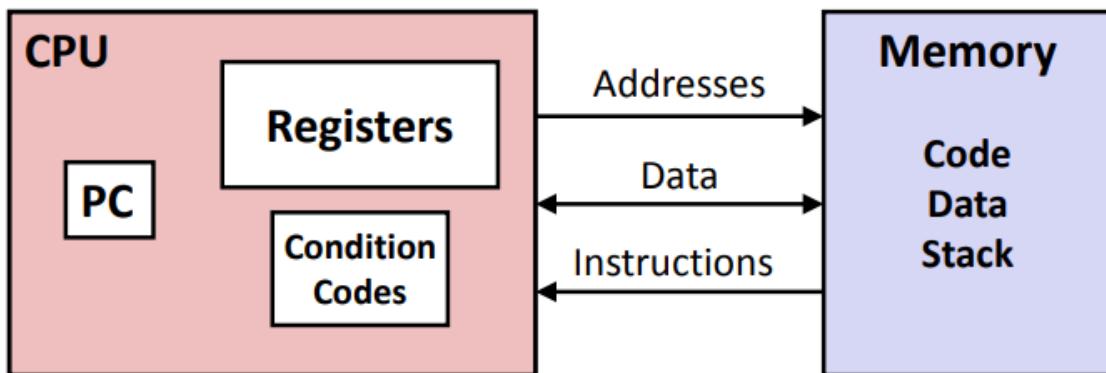
Architektura Harvardzka

architektura harwardzka

- w odróżnieniu od architektury von Neumanna, pamięć danych programu jest oddzielona od pamięci rozkazów
- Neuman vs Harvard:
 - w Neumanie CPU nie może jednocześnie czytać instrukcji i danych z pamięci. Na Harwardzkiej tak, nawet bez cache – lepszy performance.
 - Osobny przadrz dla danych i instrukcji.
- użycie:
 - small embedded computers
 - signal processing



Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called “RIP” (x86-64)
- **Register file**
 - Heavily used program data
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	movq \$0x4,%rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax,%rdx	temp2 = temp1;
	Reg	Mem	movq %rax,(%rdx)	*p = temp;
	Mem	Reg	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

Complete Memory Addressing Modes

■ Most General Form

$$D(Rb, Ri, S) \quad \text{Mem[Reg[Rb]+S*Reg[Ri]+ D]}$$

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

notacja (a, b, c) oznacza:

- w przypadku operacji mov/add/sub/...: *(a + b * c)
- w przypadku operacji lea: (a + b * c)
czyli lea nie robi dereferencji

Condition Codes (Implicit Setting)

■ Single bit registers

- | | | |
|-------------|---------------------------|--------------------------------------|
| ▪ CF | Carry Flag (for unsigned) | SF Sign Flag (for signed) |
| ▪ ZF | Zero Flag | OF Overflow Flag (for signed) |

■ Implicitly set (as side effect) of arithmetic operations

Example: **addq Src,Dest** $\leftrightarrow t = a+b$

CF set if carry out from most significant bit (unsigned overflow)

ZF set if $t == 0$

SF set if $t < 0$ (as signed)

OF set if two's-complement (signed) overflow

$(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$

■ Not set by **leaq** instruction

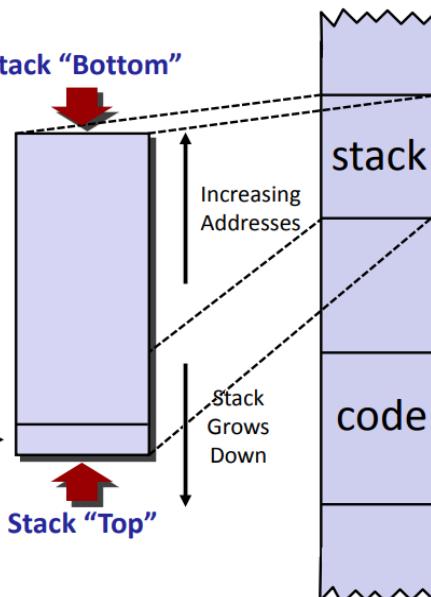
Programowanie niskopoziomowe: procedury

Stos

x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
 - address of “top” element

Stack Pointer: `%rsp` →



x86-64 Stack: Push

■ `pushq Src`

- Fetch operand at `Src`
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

Stack Pointer: `%rsp` →

Stack “Bottom”

Stack “Top”

Stack “Bottom”

Stack “Top”

Increasing Addresses

Stack Grows Down

x86-64 Stack: Pop

■ `popq Dest`

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at `Dest` (must be register)

(The memory doesn't change,
only the value of `%rsp`)

Stack Pointer: `%rsp` →

Stack “Bottom”

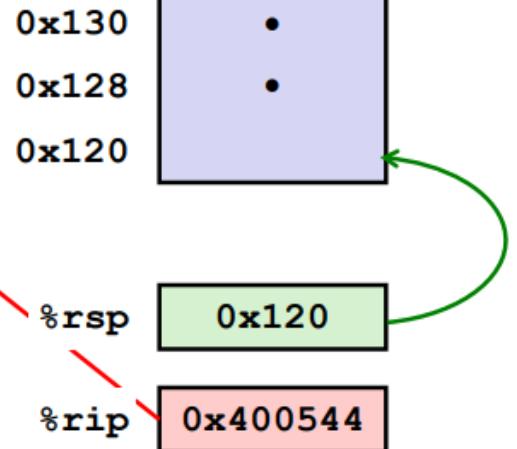
Stack “Top”

Increasing Addresses

Stack Grows Down

Control Flow Example #1

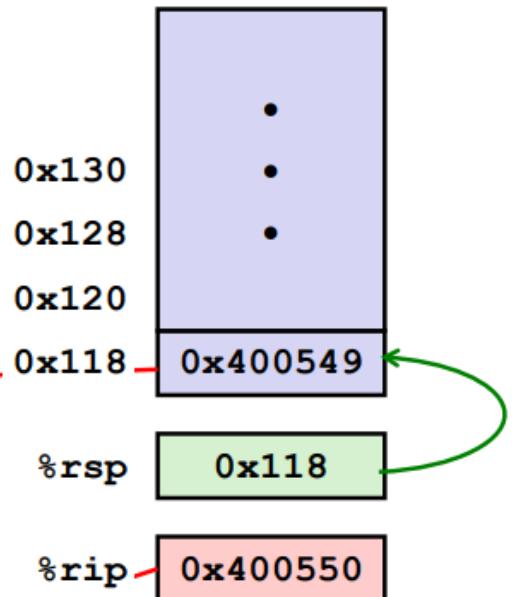
```
0000000000400540 <multstore>:  
.  
. .  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx)  
. .
```



```
0000000000400550 <mult2>:  
400550: mov %rdi,%rax  
. .  
400557: retq
```

Control Flow Example #2

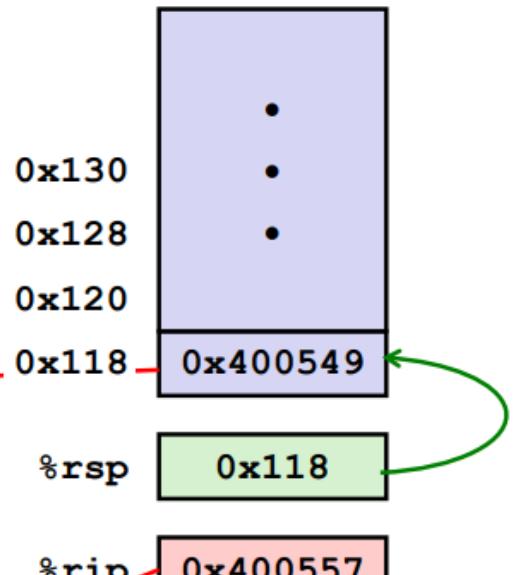
```
0000000000400540 <multstore>:  
. .  
400544: callq 400550 <mult2>  
400549: mov %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
400550: mov %rdi,%rax ←  
. .  
400557: retq
```

Control Flow Example #3

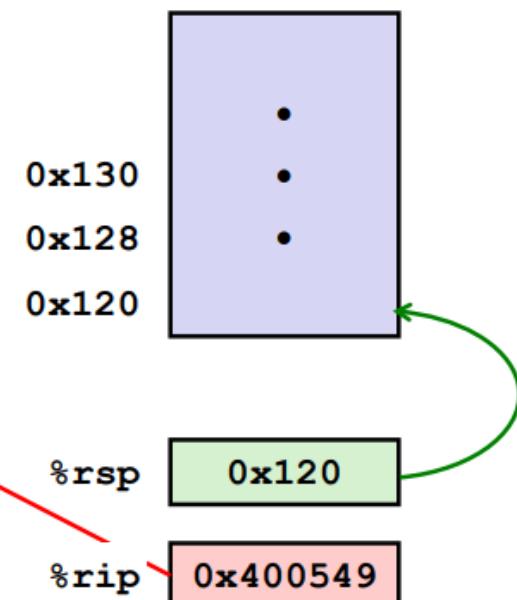
```
0000000000400540 <multstore>:  
.  
. .  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```



```
0000000000400550 <mult2>:  
400550: mov    %rdi,%rax  
. .  
400557: retq ←
```

Control Flow Example #4

```
0000000000400540 <multstore>:  
. .  
400544: callq 400550 <mult2>  
400549: mov    %rax, (%rbx) ←
```



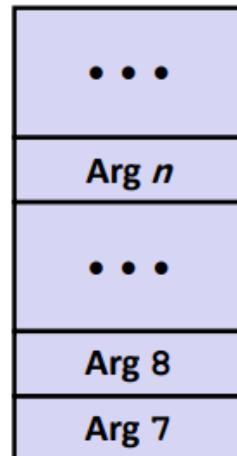
```
0000000000400550 <mult2>:  
400550: mov    %rdi,%rax  
. .  
400557: retq ←
```

Procedure Data Flow

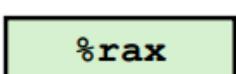
Registers

Stack

- First 6 arguments



- Return value



- Only allocate stack space when needed

Data Flow Examples

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx      # Save dest
400544: callq  400550 <mult2>  # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)    # Save at dest
...
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax      # a
400553: imul   %rsi,%rax      # a * b
# s in %rax
400557: retq               # Return
```

Stack-Based Languages

■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
 - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer

■ Stack discipline

- State for given procedure needed for limited time
 - From when called to when return
- Callee returns before caller does

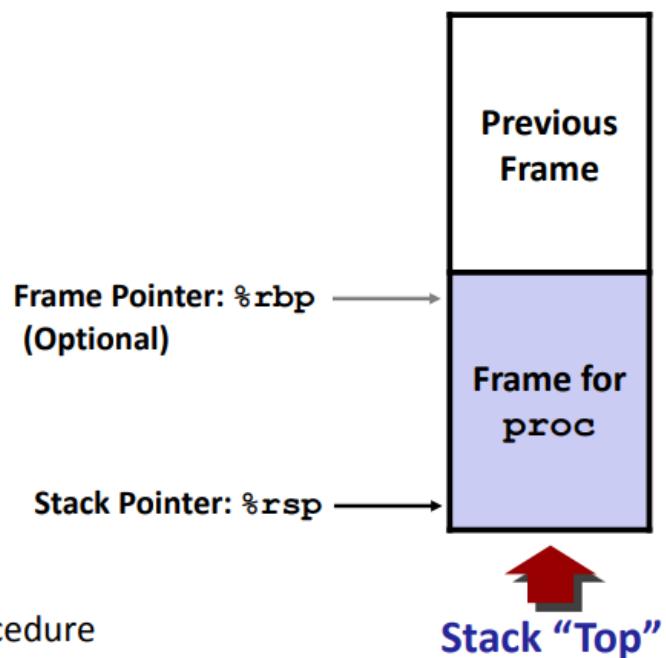
■ Stack allocated in *Frames*

- state for single procedure instantiation

Stack Frames

■ Contents

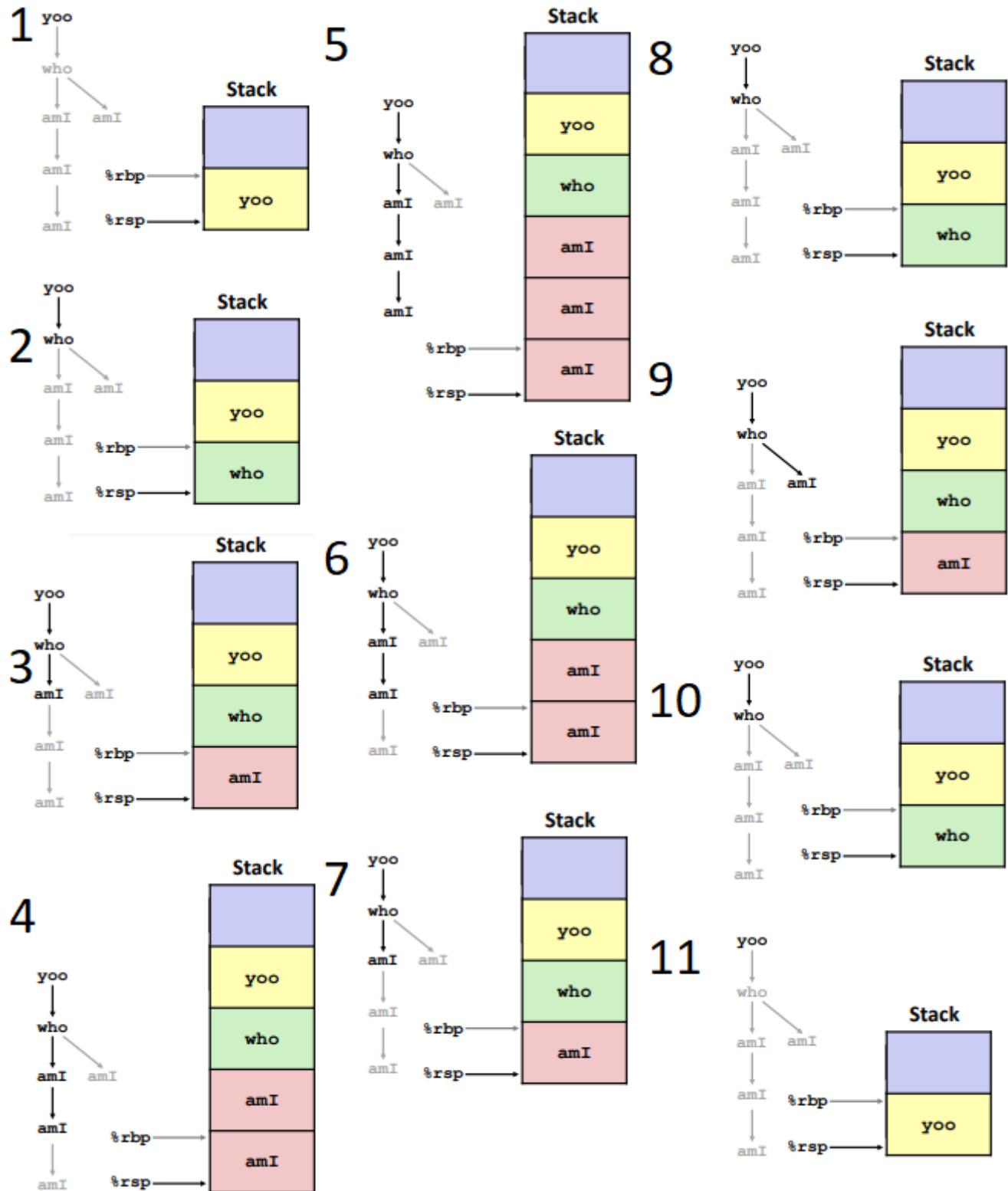
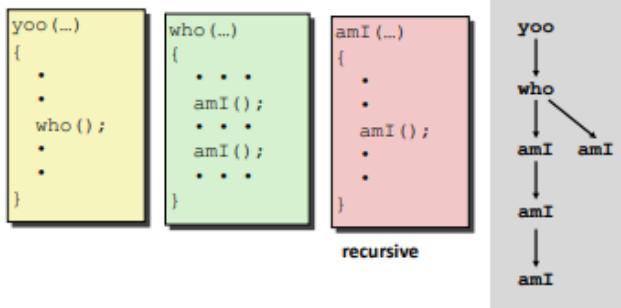
- Return information
- Local storage (if needed)
- Temporary space (if needed)



■ Management

- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by `call` instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by `ret` instruction

Call Chain Example



Example: incr

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

```
incr:
    movq    (%rdi), %rax
    addq    %rax, %rsi
    movq    %rsi, (%rdi)
    ret
```

Example: Calling incr #1

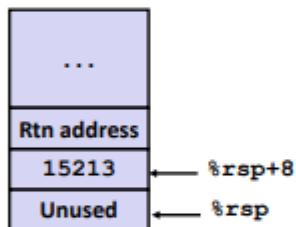
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

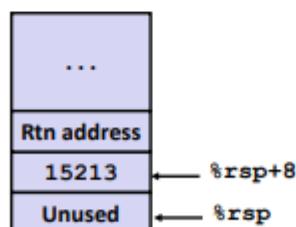
Resulting Stack Structure



Example: Calling incr #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure



```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

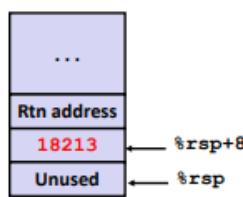
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}

call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



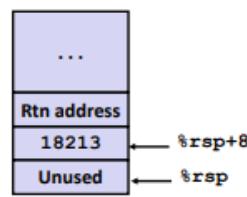
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling incr #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}

call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



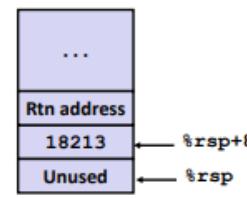
Register	Use(s)
%rax	Return value

Example: Calling incr #5a

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}

call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure



Example: Calling incr #5b

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}

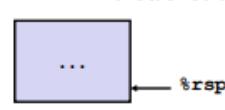
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



Register Saving Conventions

■ When procedure `yoo` calls `who`:

- `yoo` is the *caller*
- `who` is the *callee*

■ Can register be used for temporary storage?

```
yoo:  
    . . .  
    movq $15213, %rdx  
    call who  
    addq %rdx, %rax  
    . . .  
    ret
```

```
who:  
    . . .  
    subq $18213, %rdx  
    . . .  
    ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
 - Need some coordination

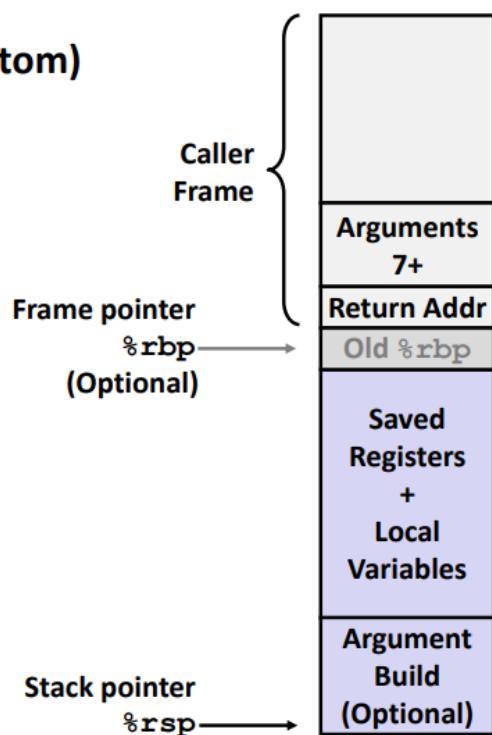
■ Conventions

- “*Caller Saved*”
 - Caller saves temporary values in its frame before the call
- “*Callee Saved*”
 - Callee saves temporary values in its frame before using
 - Callee restores them before returning to caller

x86-64/Linux Stack Frame

■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”
Parameters for function about to call
- Local variables
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)

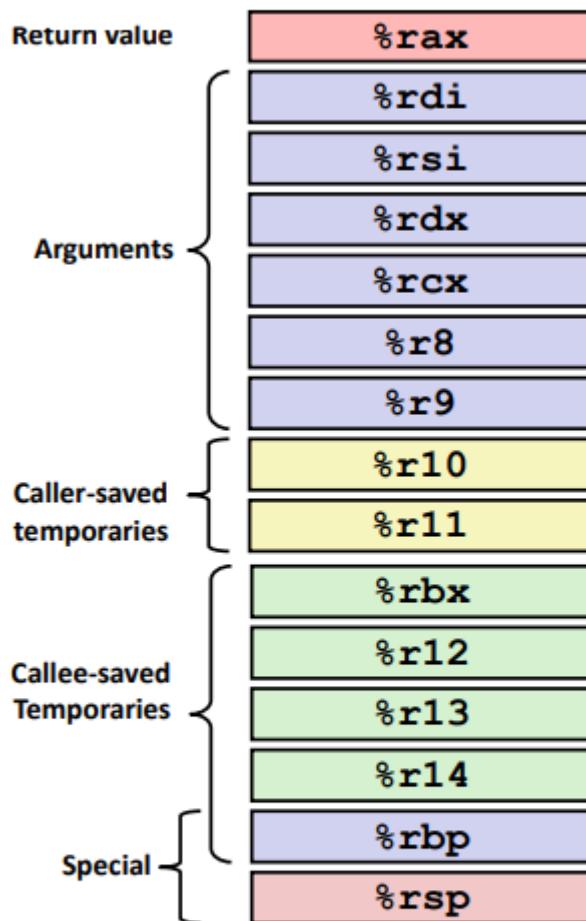


■ Caller Stack Frame

- Return address
 - Pushed by `call` instruction
- Arguments for this call

x86-64 Linux Register Usage

- **%rax**
 - Return value
 - Also caller-saved
 - Can be modified by procedure
- **%rdi, ..., %r9**
 - Arguments
 - Also caller-saved
 - Can be modified by procedure
- **%r10, %r11**
 - Caller-saved
 - Can be modified by procedure
- **%rbx, %r12, %r13, %r14**
 - Callee-saved
 - Callee must save & restore
- **%rbp**
 - Callee-saved
 - Callee must save & restore
 - May be used as frame pointer
 - Can mix & match
- **%rsp**
 - Special form of callee save
 - Restored to original value upon exit from procedure



Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq   %rdi, %rbx
    andl   $1, %ebx
    shrq   %rdi
    call    pcount_r
    addq   %rbx, %rax
    popq   %rbx
.L6:
    rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcorn */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

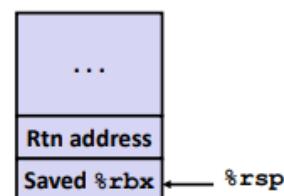
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcorn */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcorn */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



ASM -> C

Zadanie 3 (8). Przetłumacz kod procedury foobar z asemblera x86-64 do języka C. Należy posłużyć się wysokopoziomowymi instrukcjami sterującymi. Używanie etykiet i instrukcji goto jest niedozwolone.

```

1 foobar:
2     mov    %rdi, %rax
3     xor    %rdx, %rdx
4 .L2: cmp    $0, (%rax)
5     je     .L7
6     jg    .L3
7     dec    %rdx
8     jmp    .L4
9 .L3: inc    %rdx
10 .L4: add   $8, %rax
11     jmp    .L2
12 .L7: mov    %rdx, (%rsi)
13     ret

```

```

long *foobar (long *x, long *y) {
    *y = 0;
    while (*x != 0)
        if ((*x++) > 0)
            (*y)++;
        else (*y)--;
    return x;
}

```

Zadanie 4 (14). Przetłumacz kod procedury foobar z asemblera x86-64 do języka C. Należy posłużyć się wysokopoziomowymi instrukcjami sterującymi. Używanie etykiet i instrukcji goto jest niedozwolone. Opisz jednym zdaniem co robi ta procedura. Argumenty przekazywane są przez rejesty %rsi, %rdi i %rdx.

```

1 foobar:
2 .L4: mov    %rdx, %rcx
3     shr    $1, %rcx
4     lea    (%rsi,%rcx,8), %rax
5     cmp    %rdi, (%rax)
6     je     .L1
7     jge   .L5
8     lea    8(%rax), %rsi
9     dec    %rdx
10    sub   %rcx, %rdx
11    jmp   .L3
12 .L5: mov    %rcx, %rdx
13 .L3: test  %rdx, %rdx
14     jne   .L4
15     sub   %rax, %rax
16 .L1: ret

```

```

long *foobar (long key, long *tab, long size) {
    long mid = size >> 1;
    while (key != tab[mid]) {
        if (key > tab[mid]) {
            tab += mid + 1;
            size -= mid + 1;
        } else if (mid) {
            size = mid;
        } else {
            return 0;
        }
    }
    return tab + mid;
}

```

Jest to wyszukiwanie binarne, które zwraca wskaźnik na znaleziony element lub NULL w przypadku niepowodzenia.

Zadanie 5 (14). Przetłumacz procedurę o sygnaturze «long puzzle(char *, char)» z asemblera x86-64 do języka C. Należy użyć wysokopoziomowych instrukcji sterujących. Używanie etykiet i instrukcji goto jest niedozwolone. Opisz jednym zdaniem co robi ta procedura. Rejestry %sil i %dl to najmłodsze bajty %rsi i %rdx.

```
1 puzzle:  
2     leaq  1(%rdi), %rcx  
3     movq  %rdi, %r8  
4     movq  %rdi, %rax  
5 .L1:  movb  (%r8), %dl  
6     testb %dl, %dl  
7     je    .L4  
8     cmpb  %dl, (%rcx)  
9     jne   .L2  
10    cmpb  %sil, %dl  
11    je    .L3  
12 .L2:  movb  %dl, (%rax)  
13    movq  %rcx, %r8  
14    incq  %rax  
15 .L3:  incq  %rcx  
16    jmp   .L1  
17 .L4:  movb  $0, (%rax)  
18    subq  %rdi, %rax  
19    ret
```

```
1 long compress(char *start, char c) {  
2     char *dst = start;  
3     char *prev_p = start;  
4     char *curr_p = start + 1;  
5     for (; *prev_p != '\0'; curr_p++) {  
6         char prev = *prev_p;  
7         if (prev == *curr_p && prev == c)  
8             continue;  
9         *dst++ = *prev_p;  
10        prev_p = curr_p;  
11    }  
12    *dst = '\0';  
13    return dst - start;  
14 }
```

Procedura kompresuje powtarzające się wystąpienia znaku «c» w ciągu «start» pojedynczym wystąpieniem.

Zadanie 5 (12). W prostokąt poniżej wpisz treść procedury w języku C, która wykonuje to samo obliczenie, co poniższa procedura foo zaprogramowana w assemblerze. Kod w języku C może zawierać tylko instrukcje sterujące wysokiego poziomu. Przypominamy, że kodami ASCII cyfr 0...9 są liczby 48...57.

```
unsigned long foo(const char *s) {  
  
foo:  xor    %rax, %rax  
  
.L1:  movsbq (%rdi), %rdx  
      inc    %rdi  
      lea    -48(%rdx), %rcx  
      cmp    $9, %rcx  
      ja    .L2  
  
      lea    (%rax, %rax, 4), %rax  
      add    %rax, %rax  
      lea    -48(%rax, %rdx), %rax  
      jmp   .L1  
  
.L2:  ret
```

```
    unsigned long n = 0;  
    char c;  
  
    while ((c = *s++) >= '0' && c <= '9')  
        n = 10 * n + c - '0';  
    return n;
```

W prostokąt poniżej wpisz słowne wyjaśnienie, co robi funkcja foo.

Funkcja foo zamienia ciąg cyfr dziesiętnych znajdujący się na początku napisu wskazywanego przez zmienną s na liczbę bez znaku.

Zadanie 10. *Application Binary Interface*, konwencja wołania procedur.

- T ABI określa metodę wywoływanego funkcji jądra systemu operacyjnego.
- T Rejestry służące do przekazywania argumentów mogą być nadpisywane przez funkcję wołaną.
- N Jeśli wynik funkcji nie mieści się w rejestrach, to funkcja wołana musi przygotować miejsce na wynik w swojej ramce stosu.
- T Przed wywołaniem funkcji, adres wierzchołka stosu musi być wyrównany do wielokrotności rozmiaru największego słowa maszynowego.

Zadanie 9 (4). Konwencja wołania procedur.

- N Sposób wywoływanego procedur jest określony w dokumencie API (ang. *Application Programming Interface*).
- N Rekord aktywacji procedury przechowuje wszystkie jej zmienne lokalne.
- N Jeśli wynik wywołanej procedury nie mieści się w rejestrach, to jest umieszczany w jej ramce stosu.
- T Rejestry służące do przekazywania argumentów mogą być nadpisywane przez procedurę wywoływaną.

Konsolidacja i ładowanie

Co daje nam korzystanie z linkera?

- modularność – programy mogą być tworzone jako kolekcja małych plików źródłowych, a nie jako jeden wielki kod
- wydajność – osobna komplikacja; po zmianie jednego pliku źródłowego wystarczy ponownie go skompilować i jeszcze raz zlinkować całość, bez potrzeby komplikacji pozostałych, niezmienionych plików źródłowych

Co robi linker?

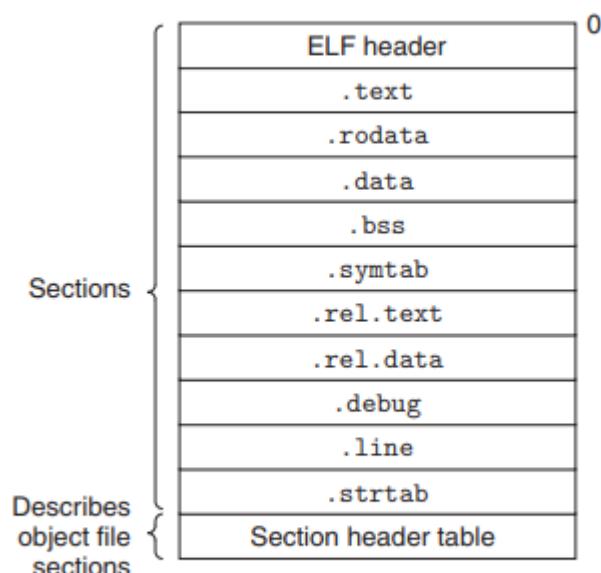
- rozwiązywanie symboli (ang. *symbol resolution*)
 - programy definiują i odwołują się do symboli (zmienne globalne i funkcje)
 - definicje symboli są przechowywane w pliku obiektowym w tablicy symboli
 - tablica symboli jest tablicą struktur
 - każdy wpis w tablicy symboli zawiera nazwę, rozmiar i lokalizację symbolu
 - w tym kroku linker kojarzy każde odwołanie do symbolu z dokładnie jedną definicją symbolu
- relokacja
 - łączy odseparowane części kodu i sekcje danych w jedne sekcje
 - relokuje symbole z ich względnych pozycji w plikach .o do końcowych bezwzględnych pozycji w pamięci w pliku wykonywalnym
 - aktualizuje wszystkie odwołania do tych symboli o nową pozycję

Rodzaje plików obiektowych

- relokowalny .o
 - zawiera kod i dane w takiej formie, że będzie można go połączyć z innym relokowalnym i utworzyć plik wykonywalny
 - każdy .o jest produkowany z dokładnie jednego .c
- wykonywalny a.out
 - zawiera kod i dane w takiej formie pozwalającej na bezpośrednie skopiowanie do pamięci, a następnie wykonanie
- dzielony .so
 - szczególny typ .o, który można załadować do pamięci i zlinkować dynamicznie w load time lub run time
 - w Windowsie to jest .dll (ang. *Dynamic Link Libraries*)

ELF binaries (ang. *Executable and Linkable Format*) – ujednolicony binarny format dla plików obiektowych .o, .out i .so

elf header	rozmiar słowa, byte ordering, typ liku (.o, exec, .so), typ maszyny (np. IA32), rozmiar i offset section header table
.text	kod maszynowy skompilowanego programu
.rodata	dane read-only, np. format stringi w printfach, tablice skoków dla switchów
.data	zainicjalizowane zmienne globalne i static lokalne (non static lokalne są na stosie)



.bss	niezainicjalizowane zmienne globalne i static lokalne; „better save space” - ta sekcja nie zajmuje miejsca w pliku obiektowym	.debug	tablica symboli ze zmiennymi lokalnymi i typedefsami, pomocne przy -g
.symtab	tablica symboli z informacjami o funkcjach i zmiennych globalnych, które zdefiniowano i do których się odwoływano w programie	.line	odwzorowanie pomiędzy numerami linii kodu w kodzie źródłowym a kodem maszynowym w .text. Jest obecna, gdy użyliśmy -g.
.rel.text	lista pozycji w .text, które trzeba będzie zmienić, gdy linker połączy ten plik obiektowy z innym. Każda instrukcja, która woła zewnętrzną funkcję lub odwołuje się do zmiennej globalnej będzie musiała być zmodyfikowana.	.strtab	tablica stringów dla tablicy symboli w .symtab i .debug i dla nazw sekcji w headerach sekcji
.rel.data	informacja o relokacji dla zmiennych globalnych zdefiniowanych lub używanych w module. Każda zmienna globalna, która jest inicjalizowana adresem zmiennej globalnej lub zewnętrznej funkcji, będzie zmodyfikowana.	section header table	offsety i rozmiary każdej sekcji

Symbole linkera

- global
 - symbole zdefiniowane w danym module, które mogą być wykorzystywane w innych modułach
 - non-static funkcje, non-static zmienne globalne
- external
 - globalne symbole używane w module, ale zdefiniowane w innym
- local
 - symbole zdefiniowane i używane wyłącznie w danym module
 - static funkcje, static zmienne globalne
 - to NIE są zmienne lokalne

Zadanie 6 (10). Wyznacz zawartość tablicy symboli, tj. zasięg widoczności (local, global) i sekcję (.text, .data, .bss, .rodata, COMMON, UNDEF) danego symbolu. Podkreś w kodzie miejsca wystąpienia relokacji.

UWAGA! W kodzie występują dwie stałe, które kompilator umieści w sekcji .rodata i przypisze im symbol.

```

1 static double array[] = {42.0, 3.14, 6.66};
2 int pow2;
3 int *pow2_p = &pow2;
4
5 static void sum(double *a, int n, double *vp) {
6   for (int i = 0; i < n; i++)
7     *vp += a[i];
8 }
9
10 void foo(void) {
11   double d = 44.2;
12   sum(array, 3, &d);
13   d = frexp(d, pow2_p);
14   printf("%f*2^%d\n", d, pow2);
15 }
```

Symbol	Zasięg	Sekcja	
array	local	.data	
pow2	global	COMMON	
pow2_p	global	.data	
foo	global	.text	
44.2	local	.data	
sum	local	.text	
frexp	global	UNDEF	
printf	global	UNDEF	
"%f..."	local	.rodata	

Zadanie 5 (10). Przeanalizuj poniższy kod i wyznacz zawartość sekcji tablicy symboli. Dla każdego symbolu podaj zasięg widoczności (local, global) oraz sekcję do której przynależy (.text, .data, .bss, .rodata, COMMON, UNDEF). Wskaż tokeny, którym zostaną przypisane wpisy w tablicy relokacji.

UWAGA: W kodzie występuje jeden niejawnny symbol, któremu nazwę przypisze kompilator.

```

1 static int array[] = {42, 1729, 0};
2 int counter;
3 int *iptr = &counter;
4
5 int incr(int *);
6 int printf(const char *, ...);
7
8 static int sum(int *a) {
9     int sum = 0;
10    while (*a)
11        sum = *a++;
12    return sum;
13 }
14
15 void foo() {
16     int n = sum(array) + incr(iptr);
17     printf("n = %d\n", n);
18 }
```

○ - miejsca wystąpienia
relokacji

SYMBOL	ZASIĘG	SEKCJA
array	local	.data
counter	global	COMMON
iptr	global	.data
incr	global	UNDEF
printf	global	UNDEF
sum	local	.text
foo	global	.text
?*	local	.rodata

(*) stała mapisowa "n = %d\n"

Zadanie 6 (10). Wyznacz zawartość tablicy symboli, tj. zasięg widoczności (local, global) i sekcję (.text, .data, .bss, .rodata, COMMON, UNDEF) danego symbolu. Podkreś w kodzie miejsca wystąpienia relokacji.

```

int sprintf(char *, const char *, ...);
void echo(const char *str);
static const double threshold = 2.1;
extern unsigned times;
void (*msg)(const char *str) = echo;

static void alert(const char *s) {
    char buf[20];
    sprintf(buf, "WARN: %s", s);
    msg(buf);
}

void check_load(double value) {
    static unsigned counter = 10;
    if (value > threshold)
        counter--;
    if (!counter) {
        alert("counter is 0");
        counter = times;
    }
}
```

Symbol	Zasięg	Sekcja
sprintf	global	UNDEF
threshold	local	.rodata
times	global	UNDEF
msg	global	.data
echo	global	UNDEF
alert	local	.text
"WARN: %s"	local	.rodata
check_load	global	.text
counter	local	.data
"counter is 0"	local	.rodata

UWAGA! W kodzie występują dwie nienazwane stałe, które kompilator umieści w sekcji .rodata i przypisze im symbol.

Krok 1: rozwiązywanie symboli

Rozwiązywanie zduplikowanych symboli – symbole mogą być albo słabe, albo silne

- słabe – niezainicjalizowane zmienne globalne
- silne – procedury i zainicjalizowane zmienne globalne

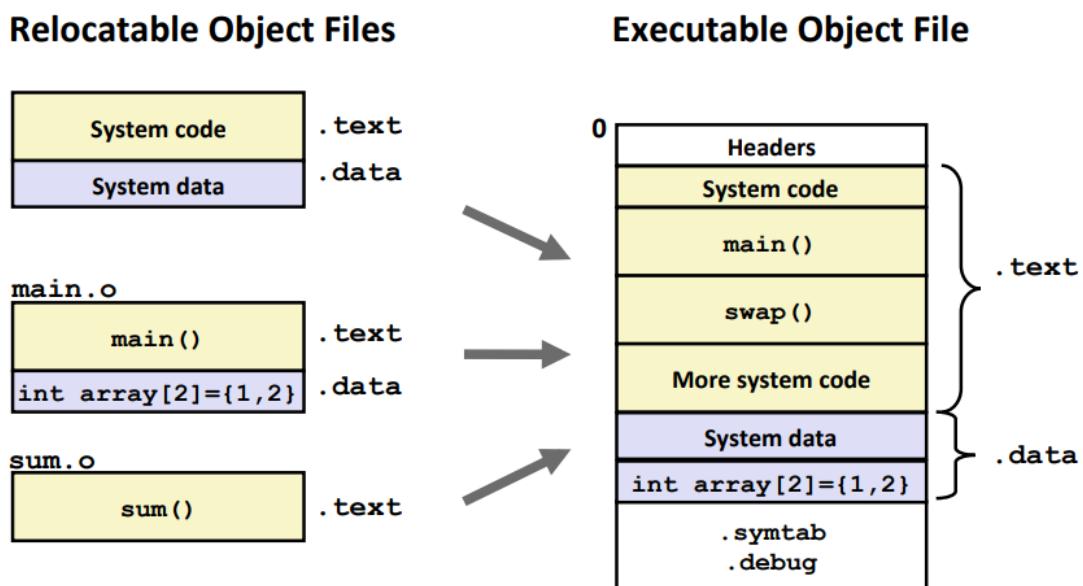
Reguły linkera

- duplikacja silnych symboli jest zakazana; linker error
- masz do wyboru silny symbol i x słabych symboli – wygra silny, odwołania do słabych dają na wyniku silny symbol
- masz do wyboru x słabych symboli – arbitralnie wybierz dowolny jeden

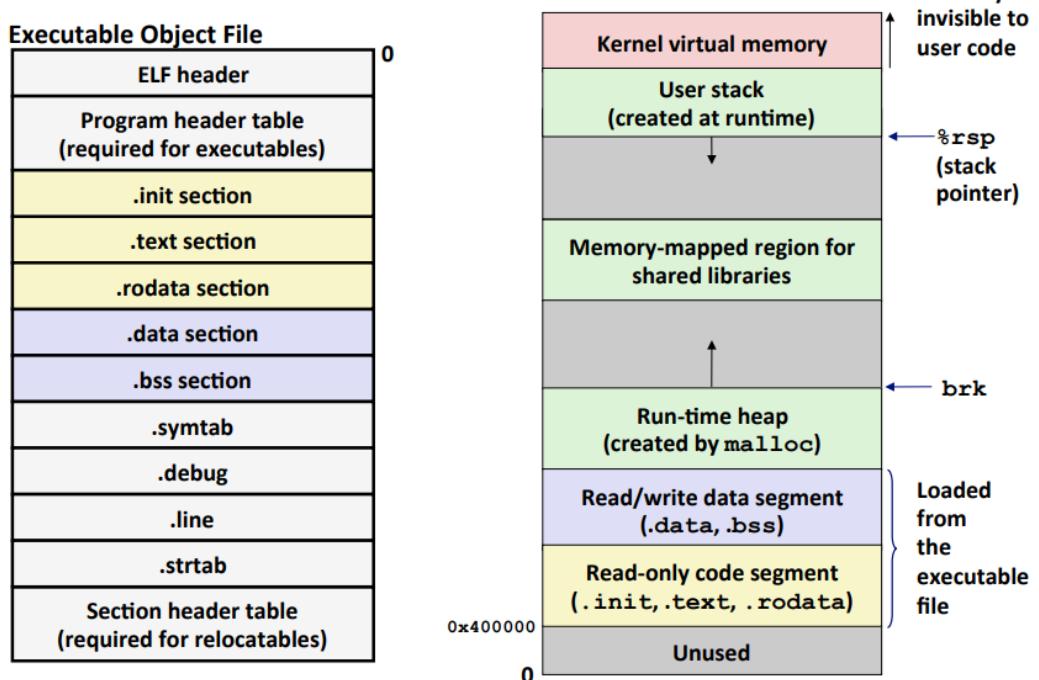
Handlowanie ze zmiennymi globalnymi

- jeśli możesz, użyj static
- inicjalizuj przy definiowaniu zmiennej
- używaj extern, odwołując się do zewnętrznej zmiennej globalnej

Krok 2: relokacja

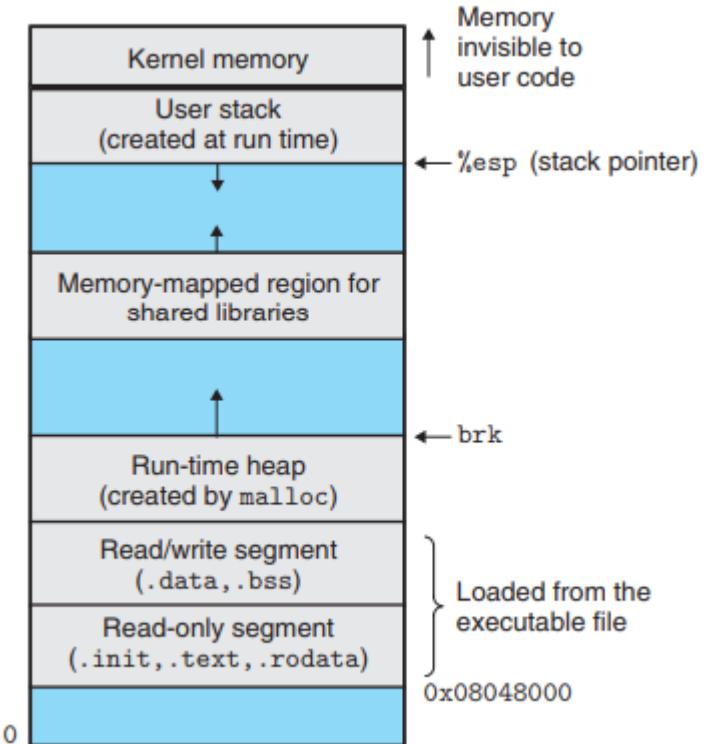


Loading Executable Object Files



Loader

1. Każdy program w uniksowym systemie uruchamia się w kontekście procesu z jego własną wirtualną przestrzenią adresową.
2. Kiedy shell uruchamia program, proces-rodzic shell robi fork i powstaje proces-dziecko jako duplikat rodzica (wywołanie systemowe fork tworzy proces potomny, który jest duplikatem rodzica).
3. Proces-dziecko wywołuje loader poprzez wywołanie systemowe execve.
4. Loader usuwa istniejące segmenty pamięci wirtualnej dziecka i tworzy nowe segmenty.
5. Nowy stos i sterta są inicjalizowane na zero. Nowe segmenty kodu i danych są inicjalizowane zawartością execa poprzez odwzorowanie stron pamięci wirtualnej na obszary pamięci (wielkości stron) execa. Proces kopiowania programu do pamięci to proces *ładowania*.
6. Ostatecznie, loader skacze do adresu _start, który wreszcie woła funkcję main.



Grupowanie często używanych funkcji

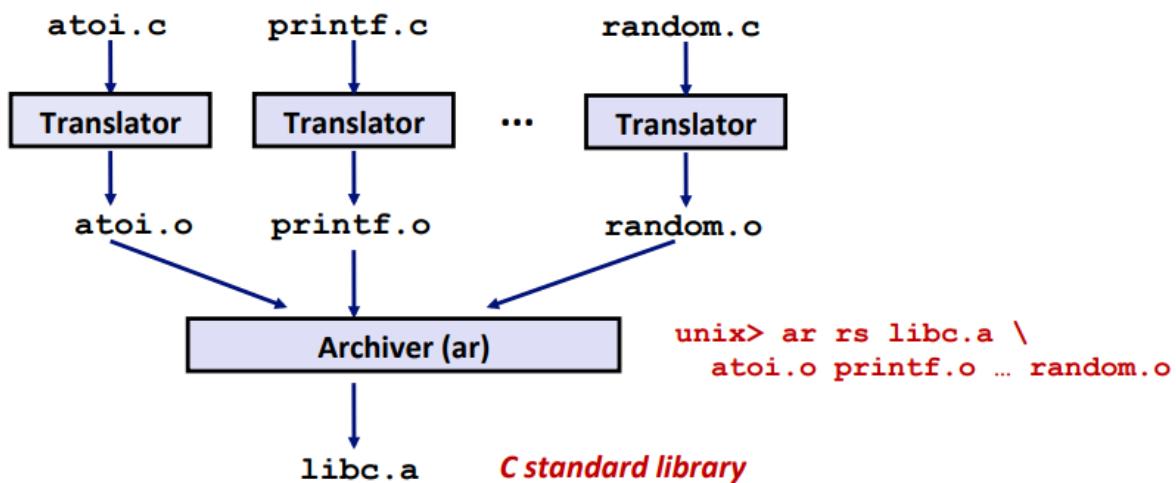
- biblioteki statyczne – pliki .a (archiwa), old-fashioned rozwiązanie
- biblioteki dzielone – nowoczesne rozwiązanie

Biblioteki statyczne

Zasada działania

- łączą powiązane obiekty relokowalne w jeden plik z indeksem (archiwum)
- linker próbuje rozwiązać nierożwiązane zewnętrzne odwołania poprzez szukanie symboli w jednym lub więcej archiwach
- jeśli w danym archiwum jest coś, co rozwiązuje odwołanie, zlinkuj to do execa
- często używane:
 - libc.a (biblioteka standardowa C) – IO, alokacja pamięci, obsługa sygnałów, liczby losowe
 - libm.a – matematyka floating point

Creating Static Libraries



<p>(a) addvec.o</p> <hr/> <pre>----- code/link/addvec.c 1 void addvec(int *x, int *y, 2 int *z, int n) 3 { 4 int i; 5 6 for (i = 0; i < n; i++) 7 z[i] = x[i] + y[i]; 8 }</pre> <hr/> <p style="text-align: right;">code/link/addvec.c</p>	<p>(b) multvec.o</p> <hr/> <pre>----- code/link/multvec.c 1 void multvec(int *x, int *y, 2 int *z, int n) 3 { 4 int i; 5 6 for (i = 0; i < n; i++) 7 z[i] = x[i] * y[i]; 8 }</pre> <hr/> <p style="text-align: right;">code/link/multvec.c</p>
---	---

Figure 7.5 Member object files in libvector.a.

<hr/> <pre>----- code/link/main2.c 1 /* main2.c */ 2 #include <stdio.h> 3 #include "vector.h" 4 5 int x[2] = {1, 2}; 6 int y[2] = {3, 4}; 7 int z[2]; 8 9 int main() 10 { 11 addvec(x, y, z, 2); 12 printf("z = [%d %d]\n", z[0], z[1]); 13 return 0; 14 }</pre> <hr/> <p style="text-align: right;">code/link/main2.c</p>	
--	--

Figure 7.6 Example program 2: This program calls member functions in the static libvector.a library.

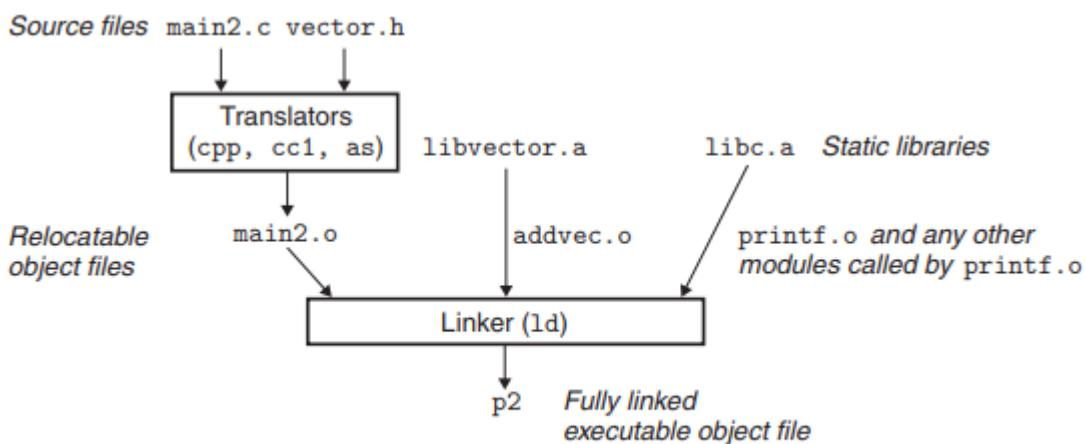


Figure 7.7 Linking with static libraries.

Algorytm linkera do rozwiązywania zewnętrznych odwołań:

- przeskanuj pliki .o i .a w porządku podanym w linii poleceń (biblioteki wrzucamy na koniec!)
- podczas skanowania utrzymuj listę jeszcze nierożwiązanych symboli
- dla każdego nowego pliku .o lub .a spróbuj rozwiązać nierożwiązane symbole
- jeśli zostały jakieś nierożwiązane symbole, wywal błąd

Wady bibliotek statycznych

- muszą być utrzymywane i co jakiś czas aktualizowane. Jeśli programista chce używać najnowszej wersji, musi jakoś się dowiedzieć, że zaszły jakieś zmiany i potem ponownie zlinkować z aktualizowaną biblioteką
- prawie każdy program w C używa funkcji IO. W runtime kod tych funkcji jest duplikowany w segmencie text dla każdego uruchomionego procesu – duże marnotrawstwo pamięci

Biblioteki dzielone

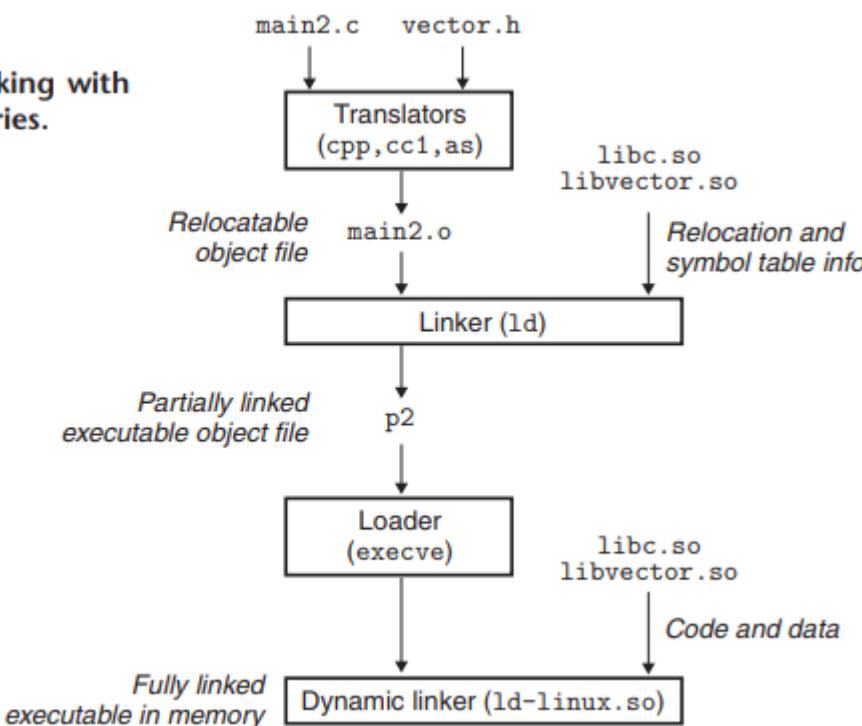
Zasada działania

- pliki obiektowe zawierające kod i dane są ładowane i linkowane dynamicznie w load time lub run time
- pliki .so lub .dll na Windowsie
- są „dzielone” na dwa sposoby
 - w danym systemie plików jest tylko jeden plik .so dla danej biblioteki. Kod i dane w tym pliku .so są dzielone przez wszystkie exeki które odwołują się do niej (w przeciwieństwie do statycznych, które są kopiowane i osadzane w exekach)
 - pojedyncza kopia sekcji .text biblioteki dzielonej w pamięci może być dzielona przez różne działające procesy (ad pamięć wirtualna)

Dynamiczna konsolidacja

- w load-time
 - gdy exec jest pierwszy raz ładowany i uruchamiany
 - często przypadek dla Linuxa, automatycznie obsługiwany przez dynamic linker (ld-linux.so)
 - standardowa biblioteka C (libc.so) jest zazwyczaj konsolidowana dynamicznie
- w run-time
 - po uruchomieniu programu
 - w Linuksie ma miejsce po wywołaniu interfejsu dlopen()

Figure 7.15
Dynamic linking with shared libraries.
(load-time)



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dlfcn.h>
4
5 int x[2] = {1, 2};
6 int y[2] = {3, 4};
7 int z[2];
8
9 int main()
10 {
11     void *handle;
12     void (*addvec)(int *, int *, int *, int);
13     char *error;
14
15     /* Dynamically load shared library that contains addvec() */
16     handle = dlopen("./libvector.so", RTLD_LAZY);
17     if (!handle) {
18         fprintf(stderr, "%s\n", dlerror());
19         exit(1);
20     }
21
22     /* Get a pointer to the addvec() function we just loaded */
23     addvec = dlsym(handle, "addvec");
24     if ((error = dlerror()) != NULL) {
25         fprintf(stderr, "%s\n", error);
26         exit(1);
27     }
28
29     /* Now we can call addvec() just like any other function */
30     addvec(x, y, z, 2);
31     printf("z = [%d %d]\n", z[0], z[1]);
32
33     /* Unload the shared library */
34     if (dlclose(handle) < 0) {
35         fprintf(stderr, "%s\n", dlerror());
36         exit(1);
37     }
38     return 0;
39 }
```

Figure 7.16 An application program that dynamically loads and links the shared library libvector.so. (run time)

Narzędzia do manipulowania plikami obiektowymi

- ar: creates static libraries, and inserts, deletes, lists, and extracts members
- strings: lists all of the printable strings contained in an object file
- strip: deletes symbol table information from an object file
- nm: lists the symbols defined in the symbol table of an object file
- size: lists the names and sizes of the sections in an object file
- readelf: displays the complete structure of an object file, including all of the information encoded in the ELF header; subsumes the functionality of size and nm
- objdump: The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the .text section
- ldd: lists the shared libraries that an executable needs at run time

Zadanie 11. Reprezentacja plików wykonywalnych, modułów i bibliotek. Konsolidacja i ładowanie.

- N Punkt wejścia (ang. *entry point*) do programu napisanego w języku C wskazuje na adres funkcji `main`.
- N Sekcja `.bss` może mieć przypisane relokacje.
- T W trakcie konsolidacji pliku wykonywalnego sekcje są łączone w segmenty.
- N Plik wykonywalny skonsolidowany dynamicznie nie posiada tablicy symboli.

Zadanie 10 (4). Konsolidacja i ładowanie.

- N Konsolidacja to proces tworzenia pliku wykonywalnego lub biblioteki ze źródeł programu.
- T Sekcja `.bss` nie jest ładowana do pamięci w trakcie uruchomiania programu.
- N Pierwsza wykonana instrukcja programu napisanego w języku C należy do procedury `main`.
- T Biblioteki współdzielone mogą być konsolidowane w momencie ładowania programu lub po jego uruchomieniu.

Model systemowy

Przerwania, wyjątki i pułapki

SO_L0.Z1. Opisz różnice między **przerwaniem sprzętowym** (ang. *hardware interrupt*), **wyjątkiem procesora** (ang. *exception*) i **pułapką** (ang. *trap*). Dla każdego z nich podaj co najmniej trzy przykłady zdarzeń, które je wyzwalają.

wyjątek - błąd przetwarzania instrukcji (nieplanowany, ale potencjalnie odwracalny)

- wystąpienie wymusza na sysopku podjęcie akcji naprawczej lub zakończenie programu
- przykłady
 - dzielenie przez zero
 - błąd strony (odwracalne)
 - arytmetyczne overflow
 - błędne odwołanie do pamięci

pulapka - generowane świadomie przez programistę z użyciem specjalnych instrukcji

- zwraca kontrolę następnej instrukcji systemowi operacyjnemu
- przykłady
 - wywołania systemowe (np. 0 - read, 1 - write, 2 - open, 3 - close, 57 - fork, 59 - execve, 62 - kill)
 - asercje (assert(0) w C)
 - breakpoint (w gdb)

przerwanie (asynchroniczny wyjątek) - generowane przez zdarzenia asynchroniczne, pochodzące z urządzeń zewnętrznych

- przykłady
 - timer systemowy
 - służy do periodycznego przełączania procesów w systemie wieloprocesorowym
 - IO z urządzeń zewnętrznych
 - wcisnięcie klawisza na klawiaturze
 - nadanie pakietu z sieci
 - zakończenie transmisji danych z dysku twardego

Mechanizm obsługi:

- rozpoznanie i identyfikacja źródła wyjątku
 - jeśli wyjątek powstał w procesorze (pułapka, wyjątek) - procesor zna dokładną przyczynę, identyfikacja zbędna
- wejście w tryb uprzywilejowany (tryb nadzorczy, *kernel mode, supervisor mode*), przerwanie wykonania strumienia instrukcji i zapamiętanie bieżącego kontekstu procesora (PC, flagi, itd.) w bezpiecznym miejscu (stos)
- załadowanie nowego kontekstu procesora i rozpoczęcie wykonywania nowego strumienia instrukcji - procedury systemowej (kontrola procesor -> sysopki) zapewniającej programową reakcję systemu na wyjątek

Obsługa przerwań

- ➊ Zachowaj kontekst programu.
- ➋ (opcjonalnie) Wykryj, które urządzenie zgłosiło przerwanie.
- ➌ Obsłuż przerwanie (odbierz dane, itp.)
- ➍ Poinformuj sprzęt, że obsłużono.
- ➎ Przywróć kontekst programu.
- ➏ Zwołaj instrukcję iret.

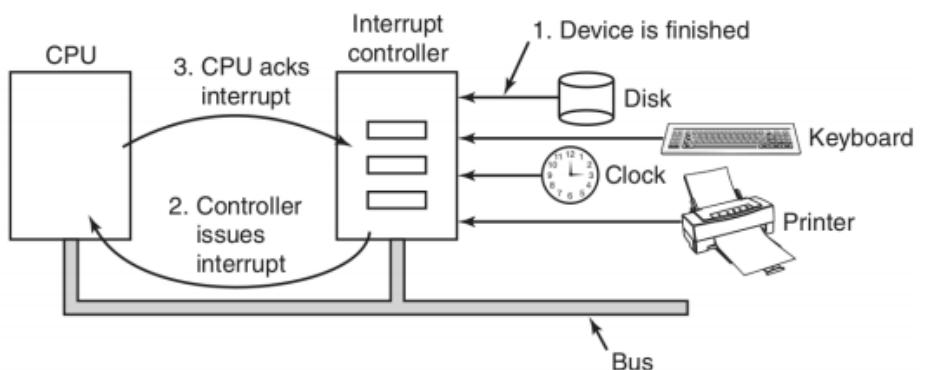
SO_L0.Z2. Opisz mechanizm **obsługi przerwań** bazujący na **wektorze przerwań** (ang. *interrupt vector table*). Co robi procesor przed pobraniem pierwszej instrukcji **procedury obsługi przerwania** (ang. *interrupt handler*) i po natrafieniu na instrukcję powrotu z przerwania? Czemu procedura obsługi przerwania powinna być wykonywana w **trybie nadzorcy** i używać odrębnego stosu?

wektor przerwań - tablica w której znajdują się adresy procedur lub też instrukcje skoku do procedur obsługi poszczególnych przerwań.

procedura obsługi przerwania -

kiedy procesorowi brakuje wyprowadzeń linii przerwań, stosuje się specjalne urządzenie - kontroler przerwań, który pośredniczy pomiędzy nim a wieloma urządzeniami żądającymi przerwania. Procedura obsługi przerwania jest sterownikiem dla kontrolera przerwań. W procedurze tej zazwyczaj odczytywany jest rejestr kontrolny kontrolera przerwań, co pozwala na stwierdzenie, które urządzenie

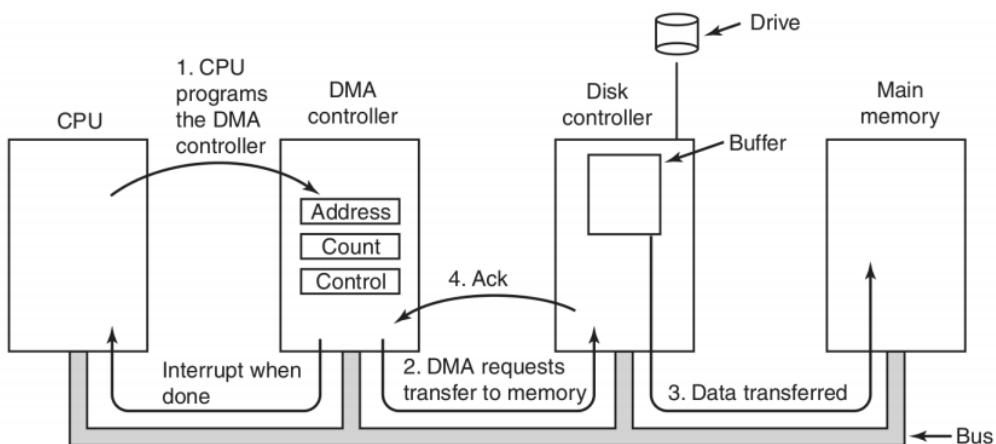
wystawiło przerwanie. Na podstawie tego sterownik kontrolera przerwań uruchamia podprogram obsługujący urządzenie, które wystawiło przerwanie.



Rysunek: Komunikacja sterowana przerwaniami

tryb nadzorcy - jest to tryb, który pozwala na wykonywanie wszystkich instrukcji procesora, włącznie z uprzywilejowanymi. Pozwala na nieograniczony dostęp do peryferiów komputera, włączanie/wyłączanie przerwań, tworzenie nowych i modyfikowanie istniejących przestrzeni adresowych. Używany jest zazwyczaj przez system operacyjny. Tryb nadzorcy pomaga zabezpieczyć dane systemu operacyjnego przed uszkodzeniem spowodowanym działaniem aplikacji.

mechanizm obsługi przerwań - kiedy procesor odbierze żądanie przerwania odkłada na stosie aktualny stan rejestru licznika programu, a następnie sprawdza w wektorze przerwań, do jakiej procedury obsługi przerwania powinien skoczyć. W procedurze tej procesor określi, jakie urządzenie zgłosiło przerwanie i wywoła odpowiednią podprocedurę. Ostatnią instrukcją w procedurze obsługi przerwania jest rozkaz reti (ang. *return from interrupt*), który pozwala mu wrócić do poprzednich czynności sprzed wystąpienia przerwania.



Rysunek: Komunikacja z użyciem DMA

Zadanie 12. Tryby pracy procesora i przerwania.

- W trakcie obsługi przerwania obsługa wyjątków jest zablokowana.
- Procesor może przejść do trybu uprzywilejowanego nie używając instrukcji uprzywilejowanych.
- Procesor odkłada na stos rejestry ogólnego przeznaczenia przed wywołaniem procedury obsługi przerwania.
- By powrócić do wykonania przerwanego programu, procedura obsługi przerwania musi wykonać specjalną instrukcję uprzywilejowaną.

Zadanie 10 (5). Podaj różnice między wyjątkiem, pułapką, a przerwaniem. Podaj przykłady zdarzeń, które je generują. Następnie opisz krótko mechanizm ich obsługi.

WYJĄTEK - błąd przetwarzania instrukcji np.: dzielenie przez zero, błąd adresowy, błędne odwołanie do pamięci. Występuje w wyniku nie zgodnego albo naprawionego lub zakończenia programu.

PUŁAPKA - generowanie świadomie przez programistę z użyciem specjalnych instrukcji. Przykład: wywołania systemowe, asynek.

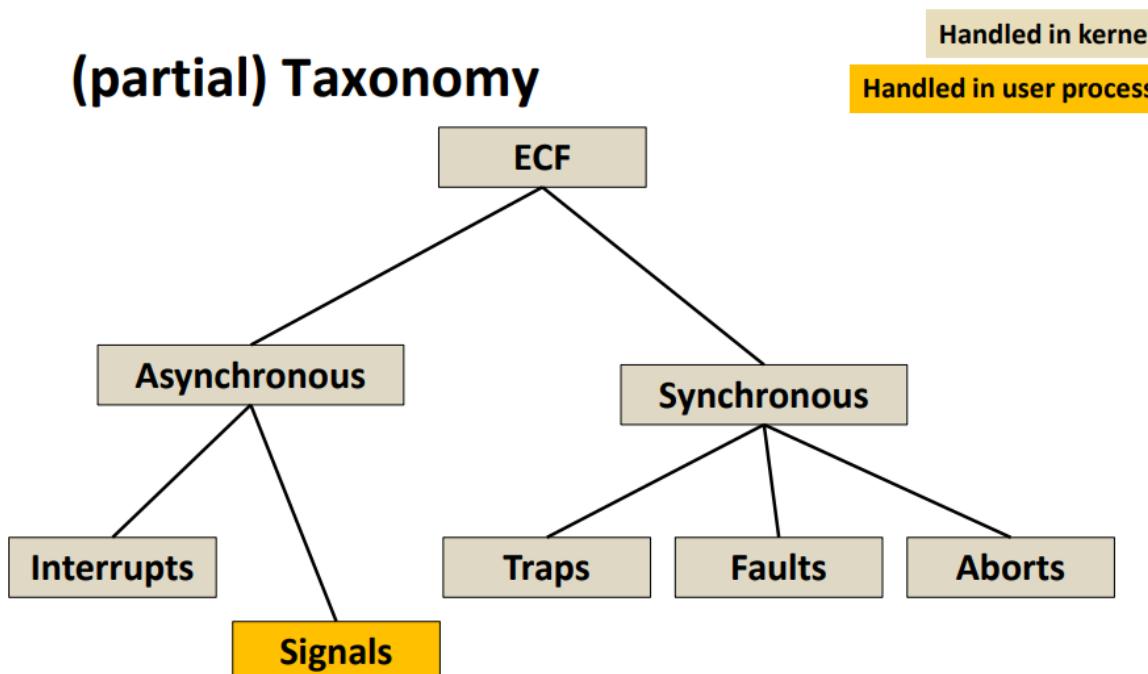
PRZERWANIE - generowane przez zdarzenia asynchroniczne, pochodzące z urządzeń zewnętrznych np.: zegar, klawiatura, dysk twardy.

Procesor przykłada w tryb uprzywilejowany, odlewa do bezpiecznego miejsca swój stan (PC, flagi, itd.) i przyjmuje przerwanie, wybiera adres procedury obsługi przerwania i przenosi tam kontrolę (do sys. op.).

Zadanie 12 (4). Tryby pracy procesora i przerwania.

- Obsługa przerwania sprzętowego może zostać przerwana w wyniku wystąpienia wyjątku procesora.
- Użycie wywołania systemowego (ang. *syscall*) powoduje przejście procesora do trybu uprzywilejowanego.
- Przed wywołaniem procedury obsługi przerwania procesor odkłada zawartość wszystkich rejestrów na stos użytkownika.
- Instrukcje uprzywilejowane mają wyższy priorytet wykonania niż inne instrukcje.

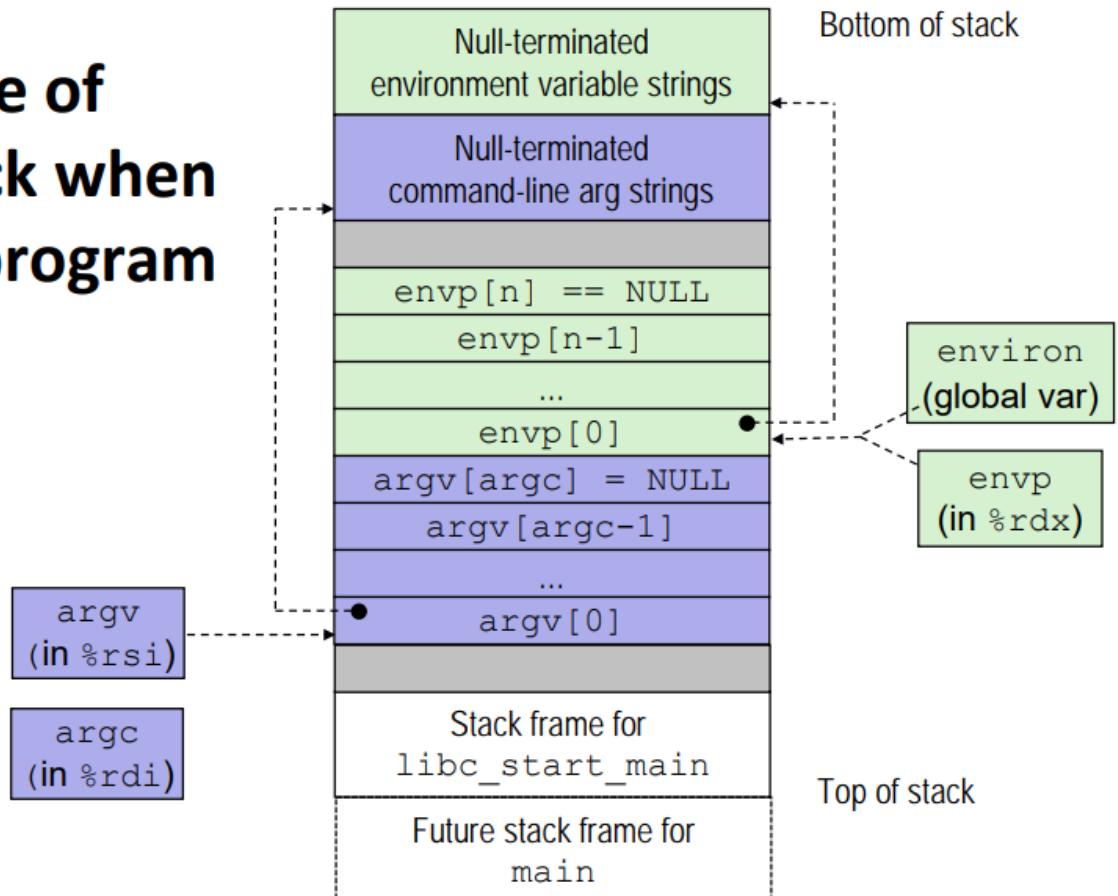
Sygnały



`execve`: Loading and Running Programs

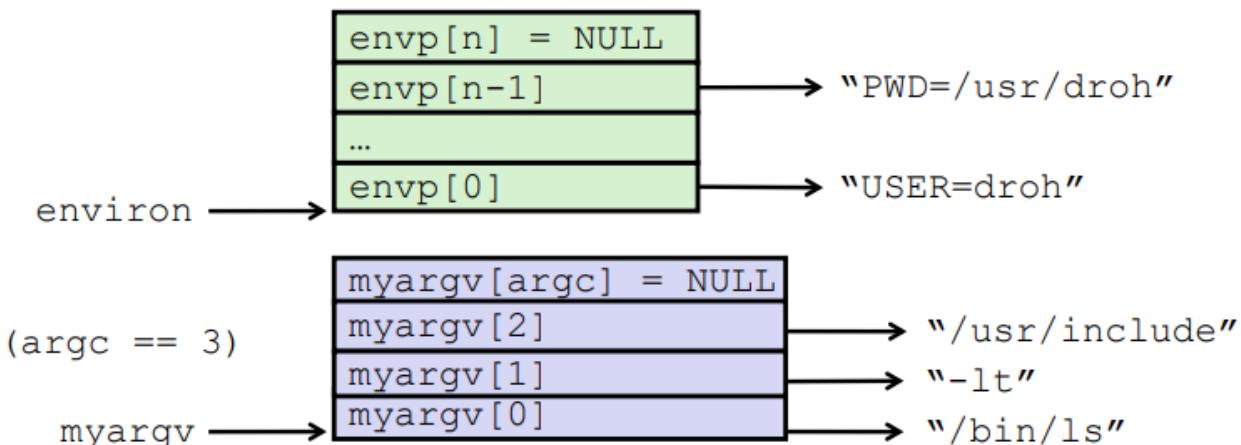
- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
 - Executable file `filename`
 - Can be object file or script file beginning with # ! interpreter (e.g., #!/bin/bash)
 - ...with argument list `argv`
 - By convention `argv[0]==filename`
 - ...and environment variable list `envp`
 - “name=value” strings (e.g., `USER=droh`)
 - `getenv`, `putenv`, `printenv`
- **Overwrites code, data, and stack**
 - Retains PID, open files and signal context
- **Called once and never returns**
 - ...except if there is an error

**Structure of
the stack when
a new program
starts**



execve Example

- Executes “/bin/ls -lt /usr/include” in child process using current environment:



```
if ((pid = Fork()) == 0) { /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system

- Akin to exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Signal type is identified by small integer ID's (1-30)
- Only information in a signal is its ID and the fact that it arrived

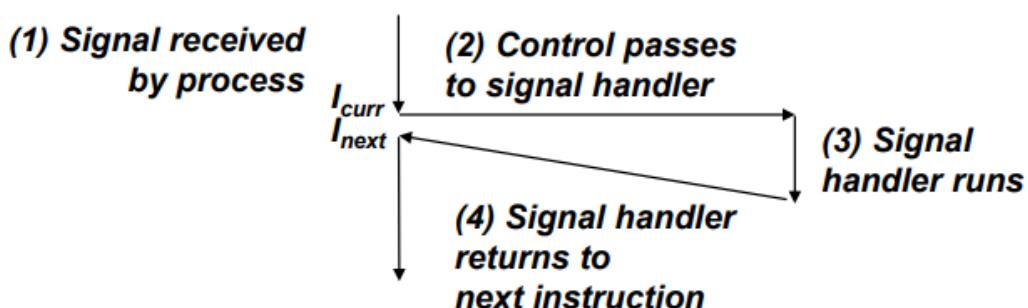
ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Concepts: Sending a Signal

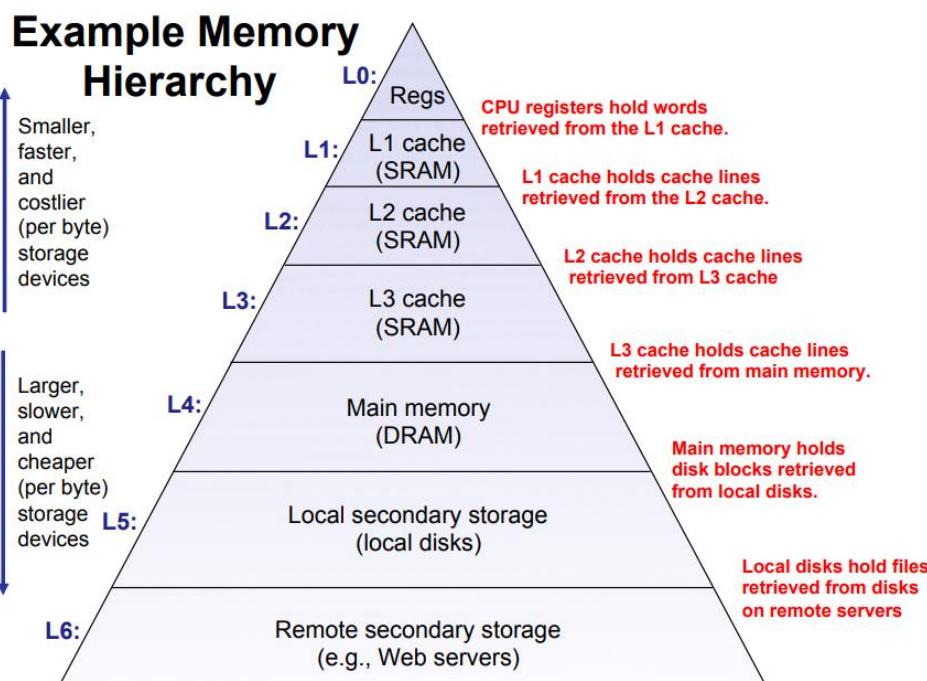
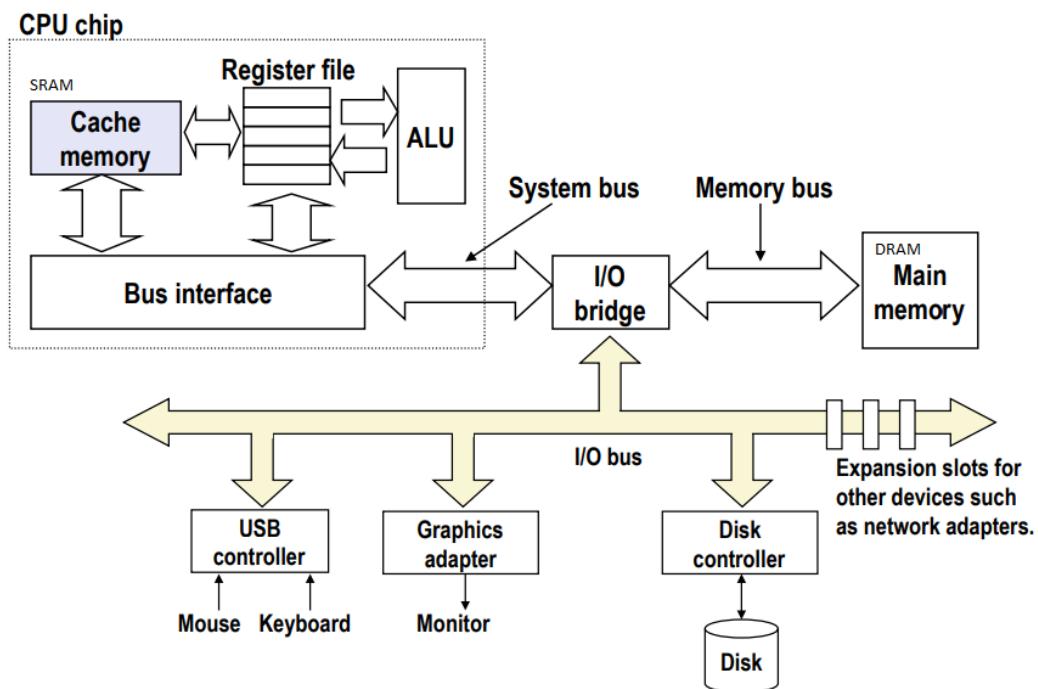
- Kernel **sends** (delivers) a signal to a **destination process** by updating some state in the context of the destination process
- Kernel sends a signal for one of the following reasons:
 - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
 - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process

Signal Concepts: Receiving a Signal

- A destination process **receives** a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Some possible ways to react:
 - **Ignore** the signal (do nothing)
 - **Terminate** the process (with optional core dump)
 - **Catch** the signal by executing a user-level function called **signal handler**
 - Akin to a hardware exception handler being called in response to an asynchronous interrupt:



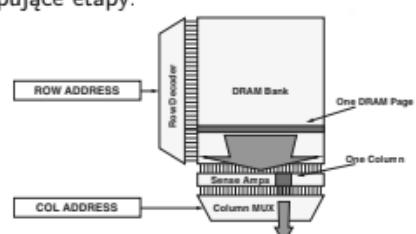
Pamięć operacyjna i dyskowa



Zadanie 11 (6). Opisz strukturę pamięci DRAM. Wymień etapy przetwarzania dostępu procesora do pamięci DDR z pominięciem pamięci podręcznej. W jakich przypadkach dostęp do pamięci jest najszybszy / najwolniejszy?

Żeby przeczytać dowolny bajt pamięci DRAM należy przejść przez następujące etapy:

- precharge: przygotowanie układów wzmacniających sygnał odczytywany z kondensatorów,
- row-select: wybór wiersza, próbkowanie ładunku i wpisanie wyników do bufora (szybka pamięć SRAM),
- column-select: wybór kolumny i wybór operacji (odczyt lub zapis)
- data-transfer: wymiana danych między procesorem, a pamięcią.



Najszybszy dostęp jest gdy wiersz jest otwarty (załadowany do bufora) i czytamy skwencyjnie (tryb *burst*). Najwolniejszy, gdy co dostęp do pamięci musimy zamykać i otwierać wiersz.

Random-Access Memory (RAM)

Kluczowe właściwości

- podstawowa jednostka to komórka (jeden bit na komórkę)
- pamięć ulotna – po wyłączeniu komputera tracimy wszystkie informacje
- dwa rodzaje:
 - static – szybsza, nie potrzebuje odświeżania, droższa
 - dynamic – wolniejsza, potrzebuje odświeżania, tańsza

Pamięci nieulotne

Typy pamięci nieulotnych

- ROM – read only memory, zaprogramowana w trakcie produkcji
- PROM – programmable ROM, może być zaprogramowana raz
- EPROM – eraseable PROM, może być wymazana
- EEPROM – electrically eraseable PROM – możliwość elektrycznego wymazania
- EEPROM – flash memory, częściowa możliwość wymazywania (d0 100k wymazywań)

Wykorzystanie pamięci nieulotnych

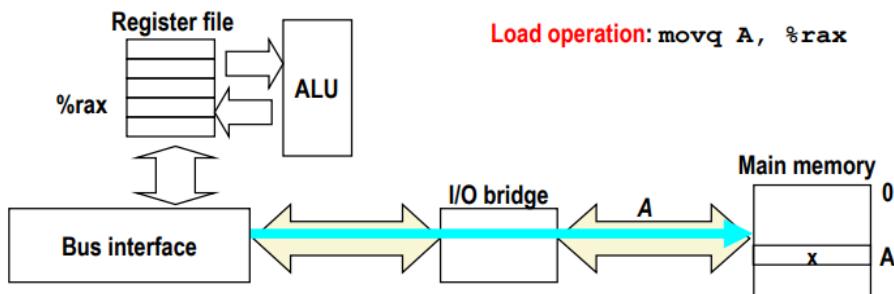
- niektóre programy są przechowywane na ROM-ach (BIOS, kontrolery dla dysków, karty sieciowe)
- SSD (smartfony, mp3, tablety, laptopy)

Szyna CPU-pamięć

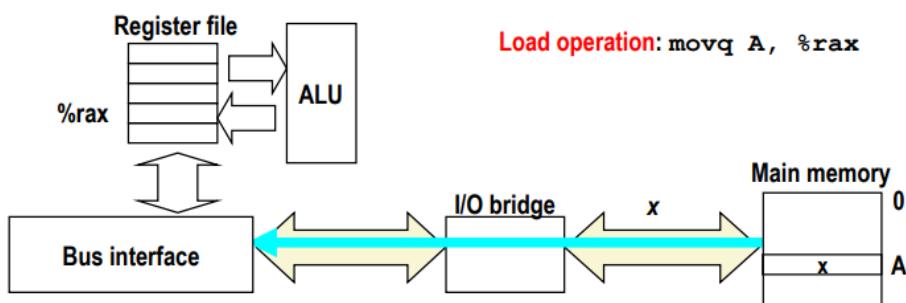
szyna - kolekcja przewodów przenoszących adresy, dane i sygnały kontrolne

Memory read transaction

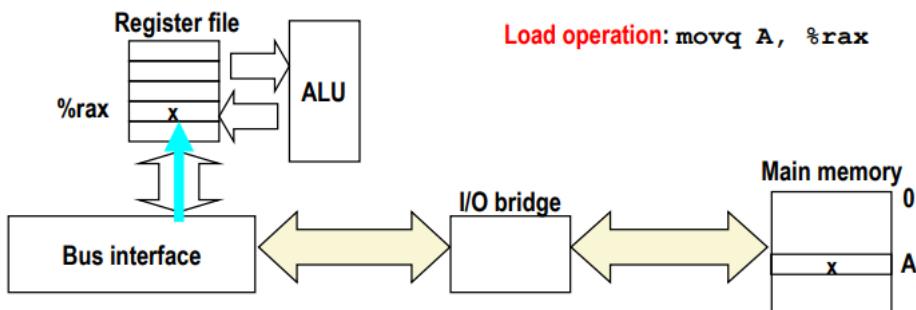
- **CPU places address A on the memory bus.**



- **Main memory reads A from the memory bus, retrieves word x, and places it on the bus.**

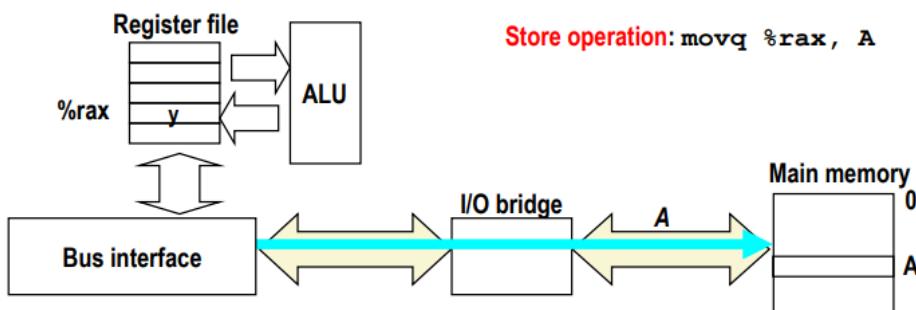


- CPU reads word x from the bus and copies it into register $\%rax$.

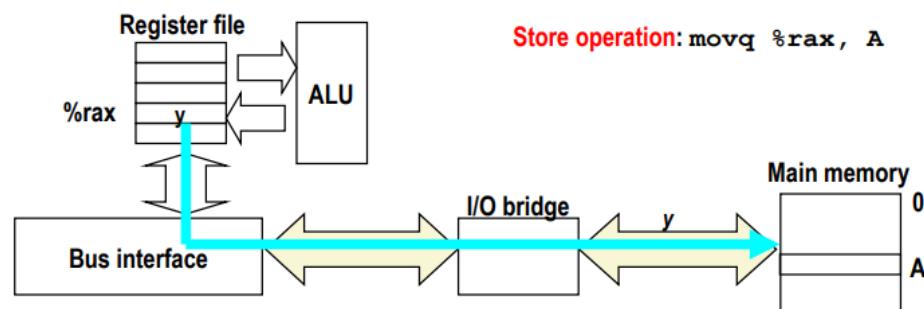


Memory write transaction

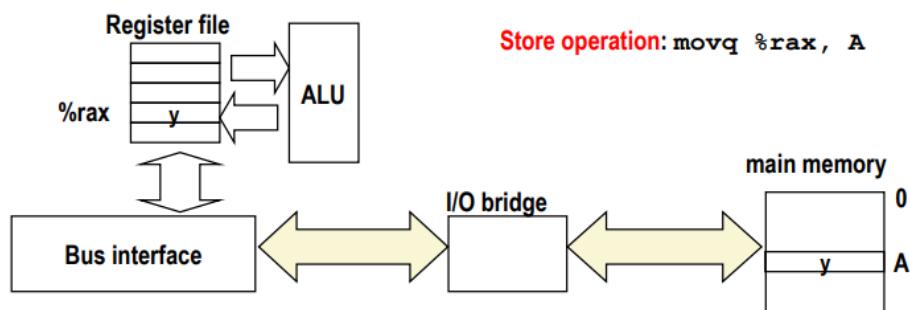
- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



- CPU places data word y on the bus.

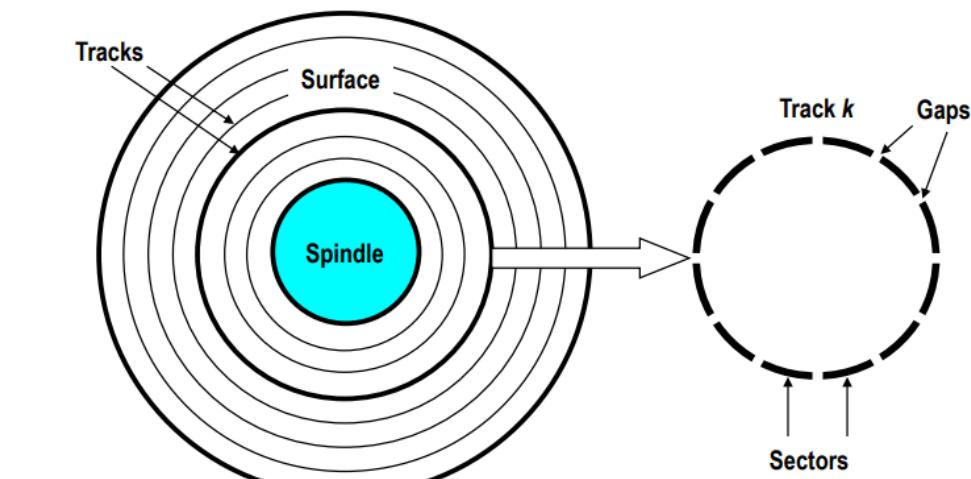


- Main memory reads data word y from the bus and stores it at address A .



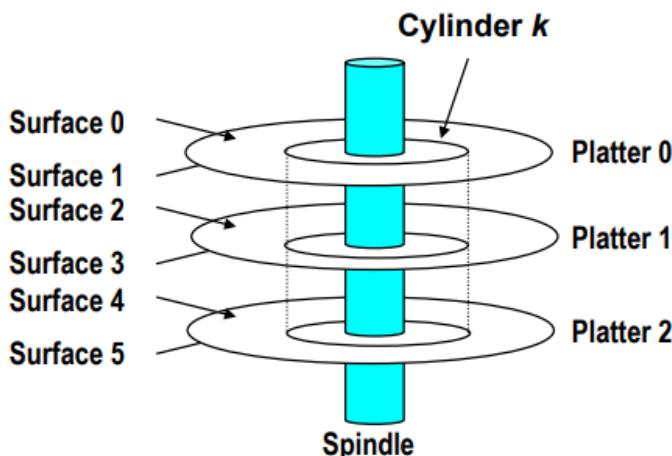
Budowa dysku

- Disks consist of **platters**, each with two **surfaces**.
- Each surface consists of concentric rings called **tracks**.
- Each track consists of **sectors** separated by **gaps**.

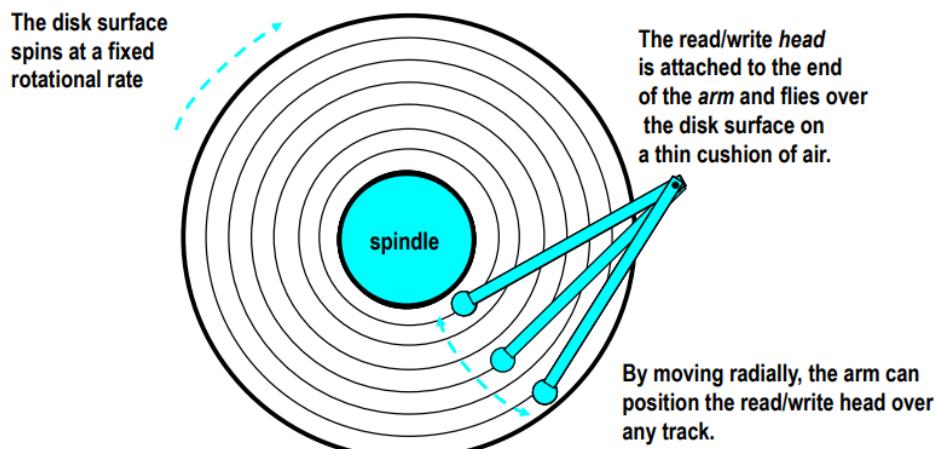


t and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

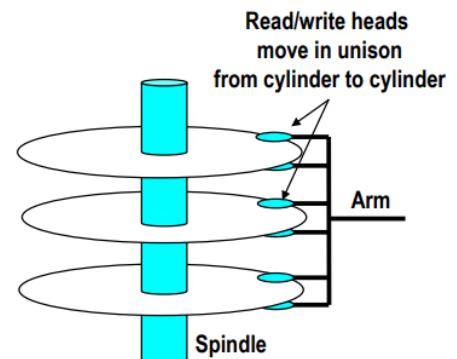
- Aligned tracks form a cylinder.



Disk Operation (Single-Platter View)

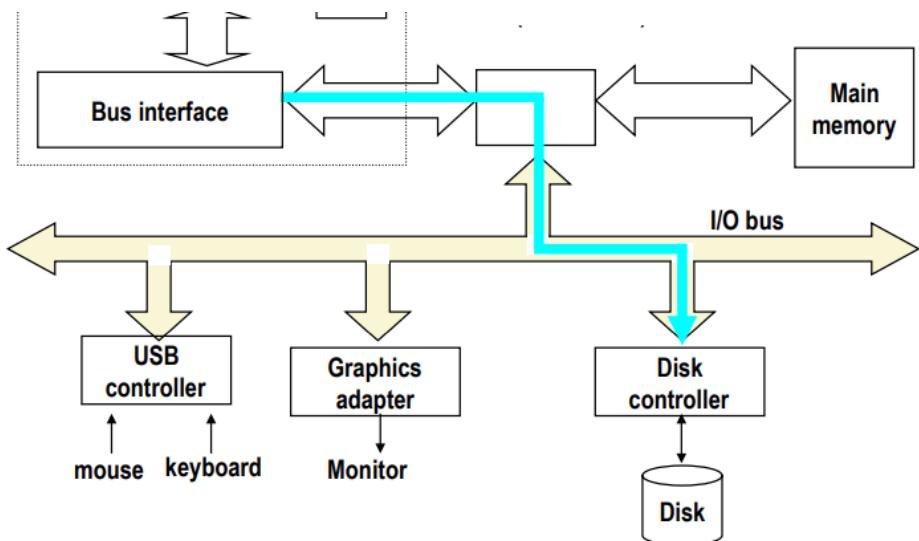


(Multi-Platter View)

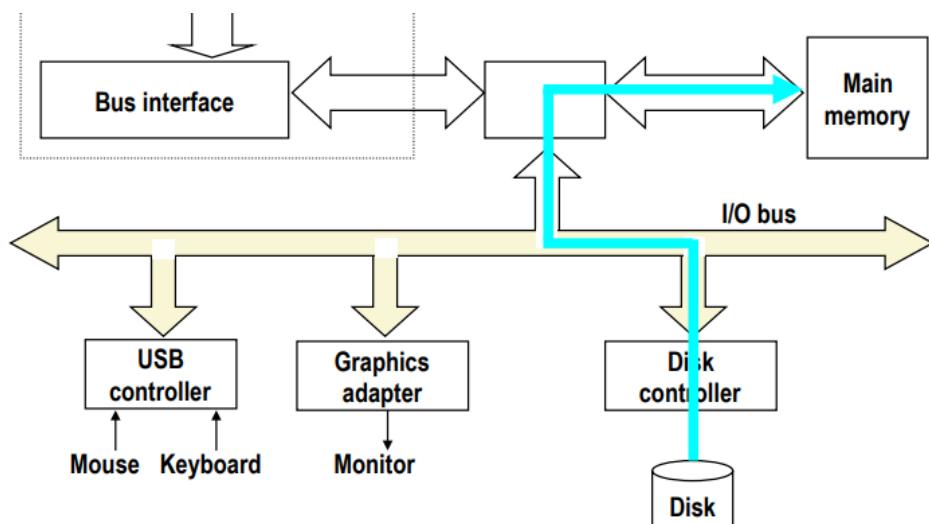


Szyna IO

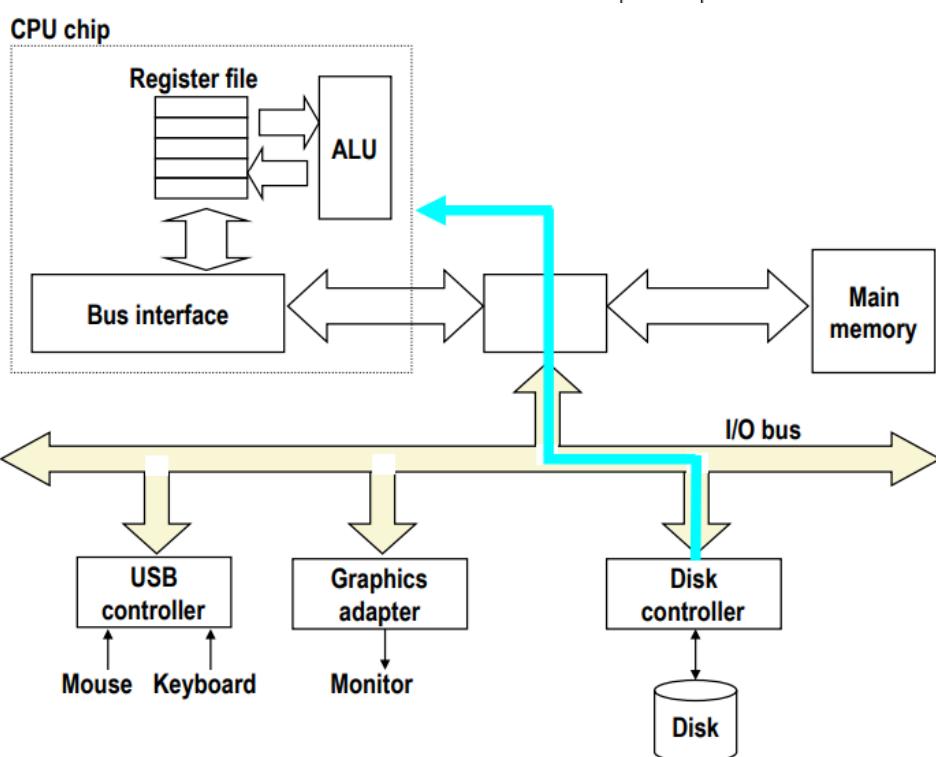
Czytanie sektora dysku



CPU inicjuje odczyt z dysku poprzez zapisanie odpowiedniej komendy z uwzględnieniem numeru logicznego bloku i docelowego adresu w pamięci do portu (adresu) skojarzonego z kontrolerem dysku.



Kontroler dysku czyta sektor na dysku i wykonuje transfer DMA (direct memory access) do pamięci głównej.



Po zakończeniu transferu DMA kontroler dysku informuje o tym CPU poprzez przerwanie.

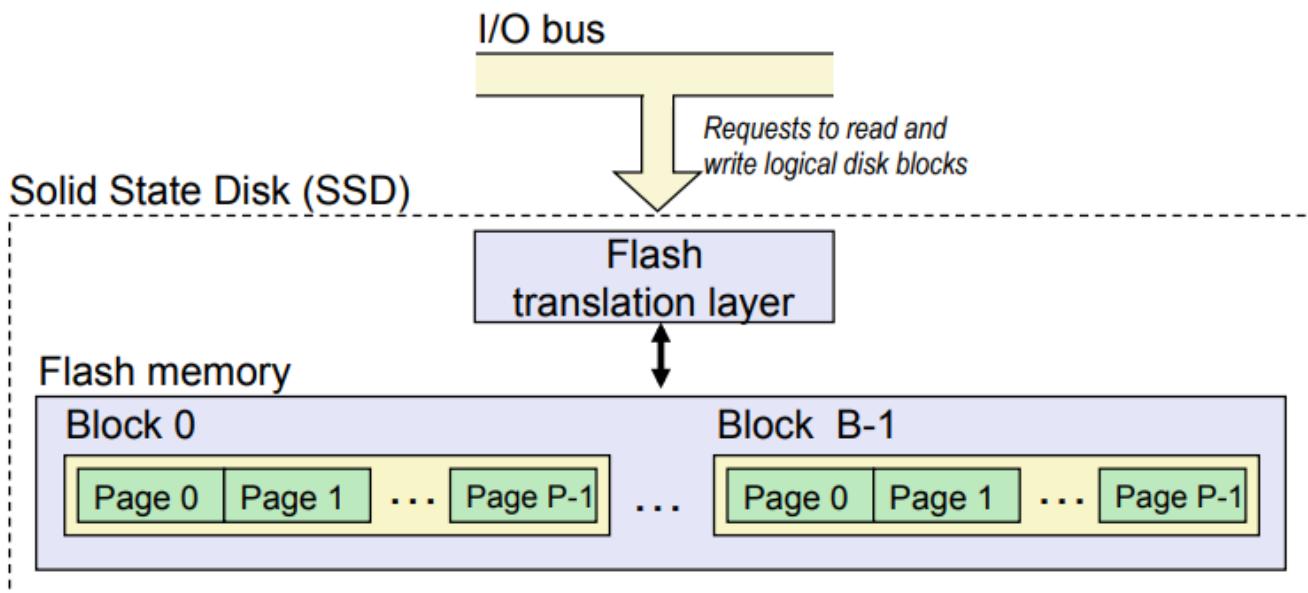
Zadanie 13. Urządzenia i obsługa wejścia-wyjścia.

- T Przerwanie może sygnalizować zdarzenia nadchodzące z więcej niż jednego urządzenia.
- T Komunikacja z użyciem przerwań jest przeważnie wydajniejsza niż odpytywanie.
- N W systemie z DMA, po zleceniu odczytu bloku pamięci operacyjnej regularnie odpytuje urządzenie i kopiuje z niego dane bez udziału procesora.
- N Wraz z odległością od środka talerza (platera) ilość sektorów na ścieżkę maleje.

Zadanie 11 (4). Obsługa wejścia-wyjścia.

- N Procesor posiada specjalne instrukcje do komunikacji z poszczególnymi typami urządzeń.
- T Wskaźniki na rejestyre urządzonych odwzorowanych w pamięć należy oznaczyć słowem kluczowym volatile.
- T Odpytywanie jest właściwą metodą komunikacji z urządzeniami, które wymagają obsługi w regularnych odstępach czasu i nie wymagają przesyłania dużej ilości danych.
- N Procesor i kontroler DMA mogą korzystać z szyny pamięci w tym samym cyklu zegarowym.

Solid State Disks (SSD)



- dane są czytane/pisane stronami (tzn. manipulowaną jednostką jest strona)
- sekwencyjny dostęp szybszy od losowego

Porównanie z ruchomymi dyskami

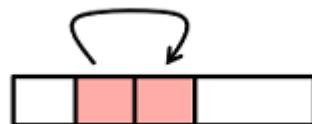
- przewagi
 - nie ma ruchomych części – szybsze, zużywają mniej energii
- wady
 - potencjalnie szybciej przestaną być zdatne do użytku (ok. 100k operacji write)
 - droższe

Lokalność

lokalność odwolań (ang. *locality of reference*) – programy mają tendencję do używania danych i instrukcji o adresach bliskich lub równych do tych, których używały ostatnio

lokalność czasowa (ang. *temporal locality*) - jeśli dane miejsce w pamięci zostało użyte, to prawdopodobnie zostanie użyte ponownie w bliskiej przyszłości

lokalność przestrzenna (ang. *spatial locality*) - jeśli obiekt był ostatnio żądanego, to prawdopodobnie będą też żądane obiekty leżące blisko (adres maszynowy) niego i warto z zachować obszar pamięci “dookoła”.



Przykład

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

data references

- sukcesywne dostępy do kolejnych elementów tablicy (co 1 krok) – przestrzenna
- dostęp do zmiennej sum w każdej iteracji – czasowa

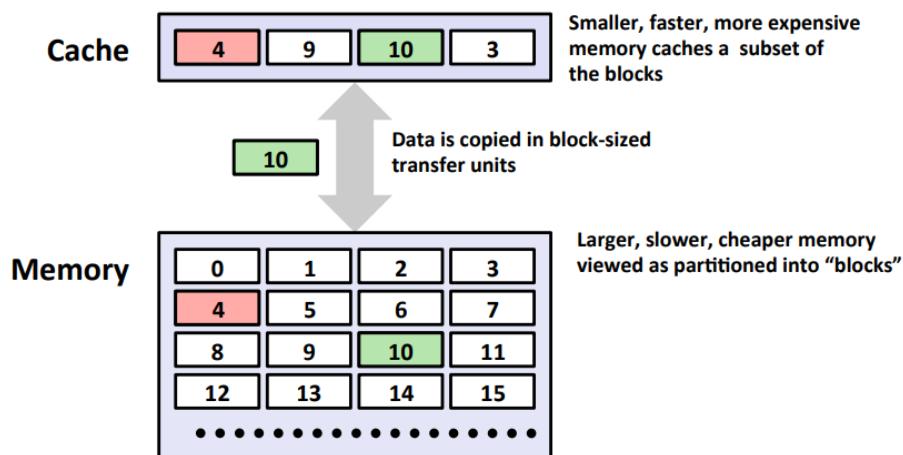
instruction references

- instrukcje sekwencyjne (nie ma goto itp.) – przestrzenna
- powtarzalna instrukcja w pętli – czasowa

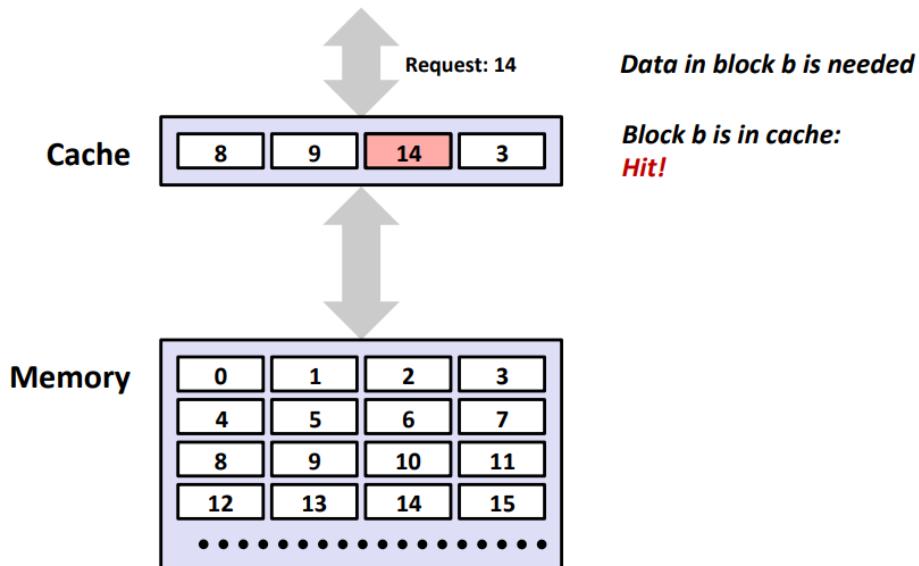
Zadanie 14. Lokalność i interakcja z pamięcią podręczną.

- [N] Przeglądanie listy dwukierunkowej charakteryzuje się dobrą lokalnością przestrzenną.
- [N] Proces generuje dużo błędów stron, jeśli jego zbiór rezydentny jest większy od zbioru roboczego.
- [T] Przetwarzanie kodu pętli wykazuje dobrą lokalność czasową.
- [N] Pamięć podręczna z polityką zapisu *write-through* może dawać niepożądane efekty przy zapisie do pamięci, która będzie kopiowana z użyciem DMA.

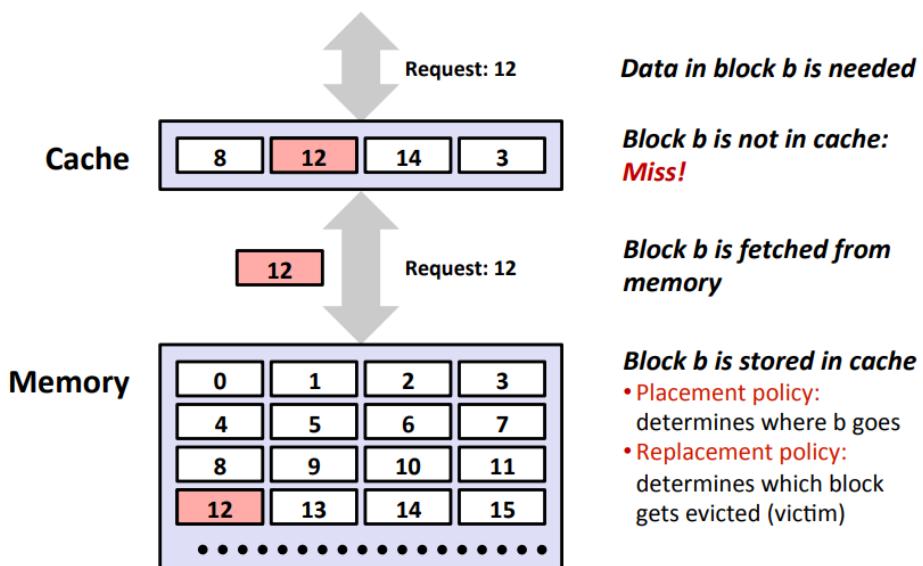
General Cache Concepts



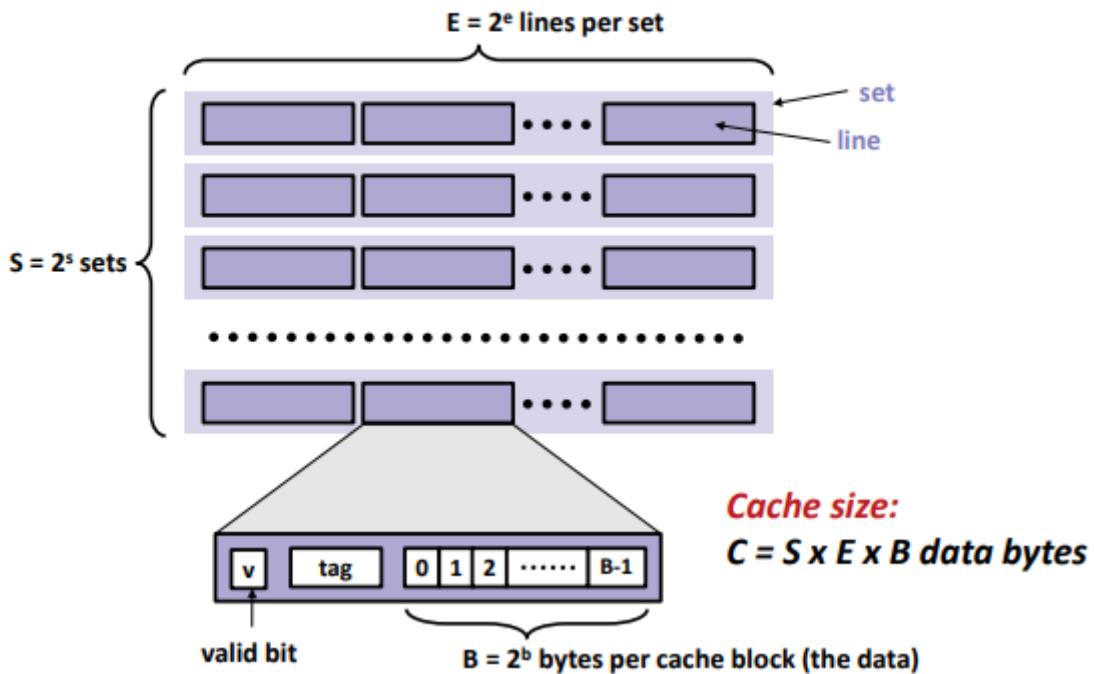
General Cache Concepts: Hit



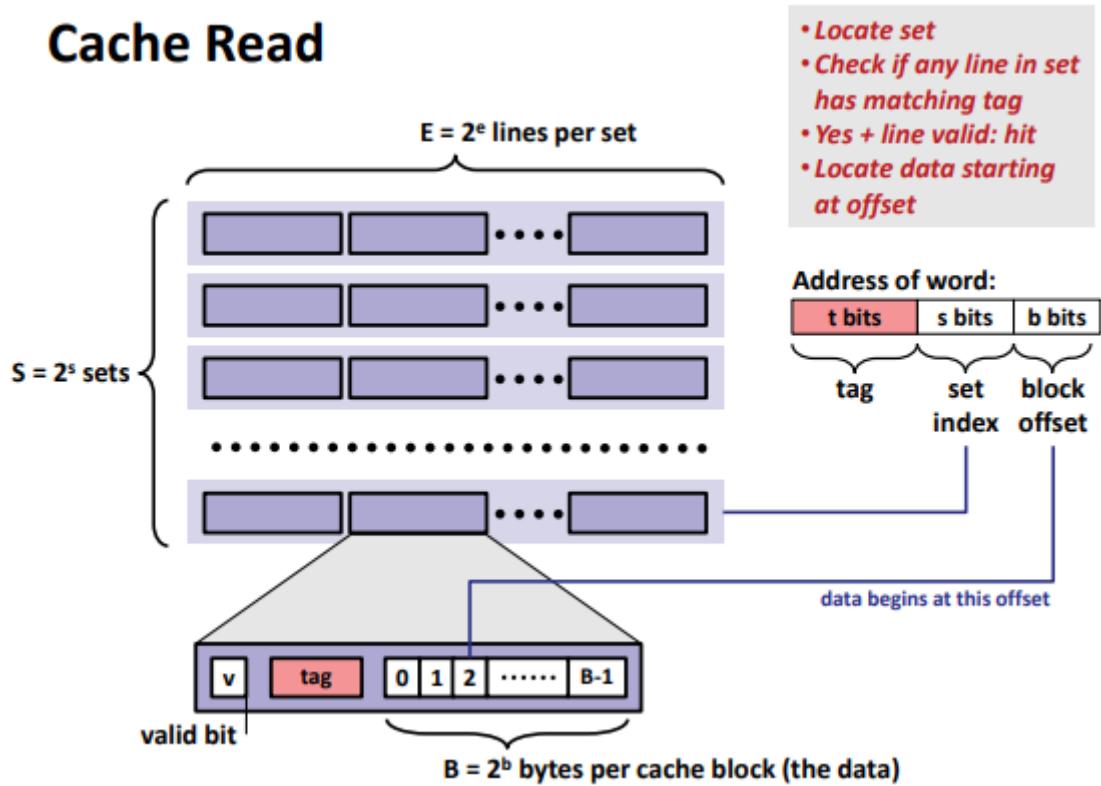
General Cache Concepts: Miss



General Cache Organization (S, E, B)

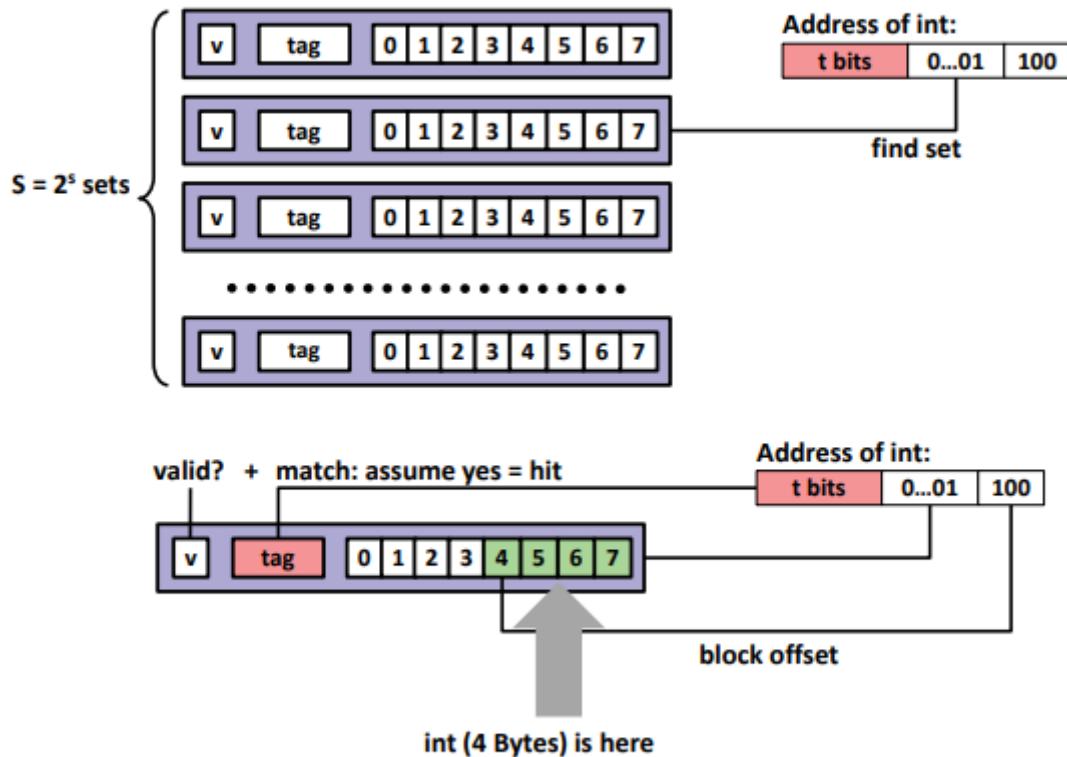


Cache Read



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

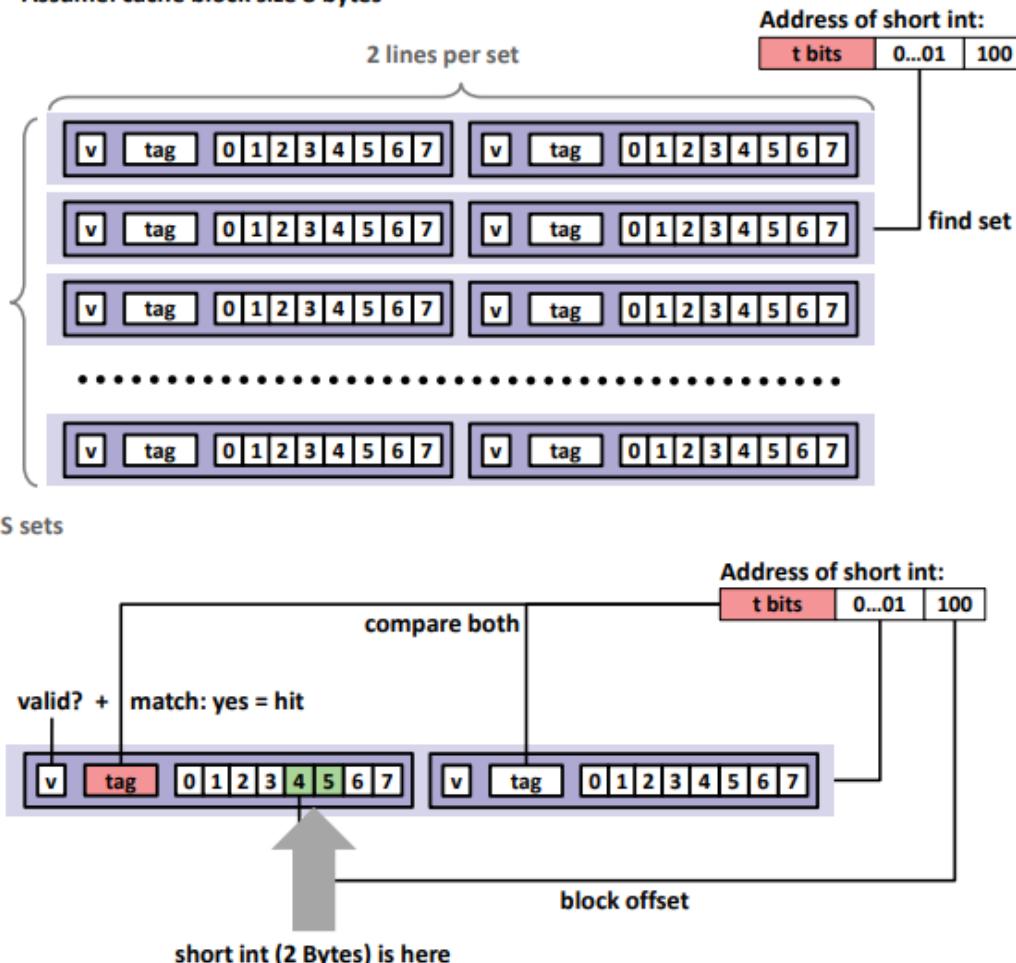


If tag doesn't match: old line is evicted and replaced

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



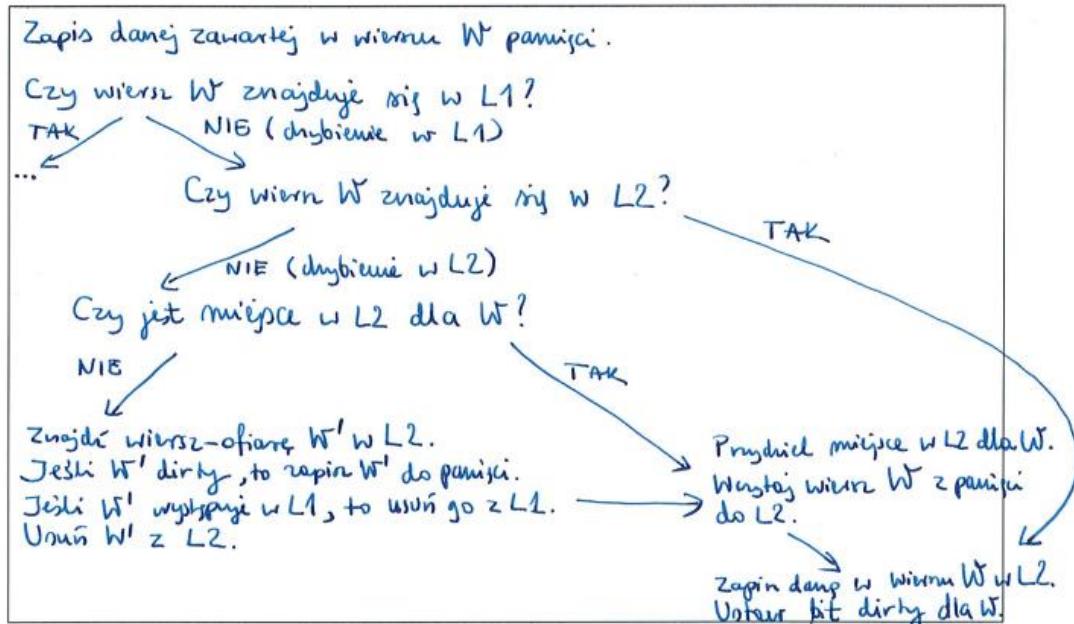
No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

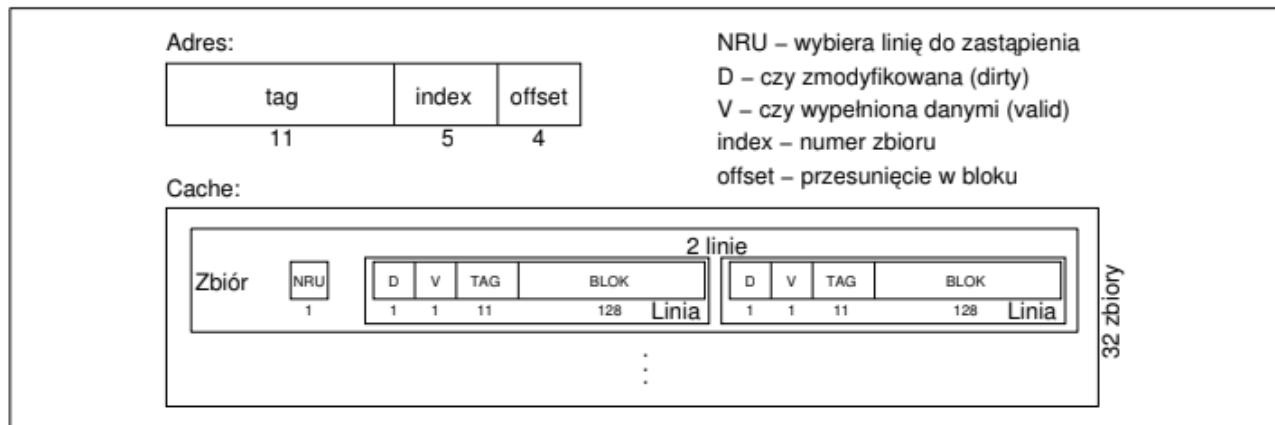
What about writes?

- Multiple copies of data exist:
 - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
 - Write-through (write immediately to memory)
 - Write-back (defer write to memory until replacement of line)
 - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
 - Write-allocate (load into cache, update line in cache)
 - Good if more writes to the location follow
 - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
 - Write-through + No-write-allocate
 - Write-back + Write-allocate

Zadanie 6 (12). Rozważmy system z dwupięciomową pamięcią podręczną z zawieraniem, tzn. $L_1 \subseteq L_2$. Pamięć L1 implementuje politykę zapisu *write-through* z *write-no-allocate*, a L2 *write-back* z *write-allocate*. Z użyciem drzewa decyzyjnego przedstaw jakie kroki należy wykonać, by obsłużyć chybienie przy zapisie do L1. Nie zapomnij rozważyć przypadku, gdy pamięć podręczna jest całkowicie wypełniona!



Zadanie 7 (6). Mamy następującą charakterystykę pamięci podręcznej: adresy 20-bitowe, łącznie 64 linie, bloki 16 bajtowe; organizacja dwudrożna sekcyjno-skojarzeniowa; polityka zapisu: *write-back*, zastępowania: *ostatnio nie używany*. Podaj format adresu, organizację sekcji i linii. Opisz metadane bloku, ich rozmiar oraz przeznaczenie.



Zadanie 4 (8). Rozważmy sekcyjno-skojarzeniową pamięć podręczną danych o rozmiarze 32 KiB. Zbiory zawierają po 8 linii. Rozmiar bloku wynosi 64 bajty. Wymiana odbywa się zgodnie z algorytmem LRU. Jednostką adresowania jest bajt. Rozważmy następujący program:

```
#define ARRAY_SIZE 4100
uint64_t A[ARRAY_SIZE], s = 0;

for (int i = 0; i < 10; i++)
    for (int j = 0; j < ARRAY_SIZE; j++)
        s ^= A[j] * i;
```

Tablica A jest umieszczona w pamięci pod adresem 0x80000. Przed wykonaniem programu pamięć podręczna danych jest pusta. Obliczenia na zmiennych i , j i s nie generują dostępu do pamięci. Niech *chybienie zastępujące* ma miejsce wówczas, gdy jest związane z usunięciem ofiary ze zbioru, a *niezastępujące*, gdy blok zostaje skopiowany do nieużywanej linii pamięci podręcznej. Uzupełnij zdania: W pierwszej iteracji zewnętrznej pętli liczba chybień niezastępujących wynosi

pujących wynosi 512, a zastępujących — 1. W drugiej iteracji zewnętrznej

pętli liczba chybień niezastępujących wynosi 0, a zastępujących — 9.

Łączna liczba chybień podczas pracy algorytmu wynosi 594.

Pamięć wirtualna

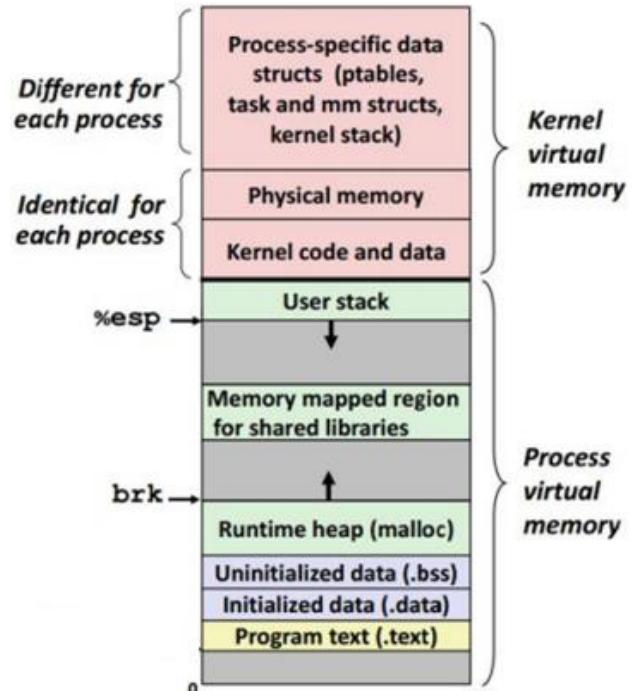
Przestrzeń adresowa

Przestrzeń adresowa

- to zbiór adresów, które proces może wykorzystać do zaadresowania pamięci. Każdy proces ma swojego własnego przadra, który jest niezależny od przadrów należących do innych procesów (poza sytuacjami, gdy procesy chcą współdzielić swoje przadry)
- każdy proces systemu Linux dysponuje przadrem podzielonym na trzy logiczne segmenty: tekst, dane i stos

Przadr procesu

- tekst
 - obejmuje rozkazy maszynowe składające się na kod wykonywalny programu
 - jest generowany przez kompilator i assembler, które tłumaczą kod C na kod maszynowy
 - zwykle jest dostępny tylko do odczytu
 - nie zmienia swojego rozmiaru ani treści w żaden sposób
- dane
 - miejsce składowania wszystkich zmiennych, łańcuchów, tablic i innych danych programu
 - składa się z dwóch części:
 - dane inicjalizowane - zmienne i stałe kompilatora, które w czasie uruchamiania programu wymagają przypisania konkretnych wartości początkowych
 - danych nieinicjalizowanych BSS - zmienne te są inicjalizowane wartością zero po załadowaniu programu
 - segment może być modyfikowany – programy stale modyfikują swoje zmienne, mogą chcieć dynamicznie przydzielać pamięć w trakcie wykonania
- stos
 - rozpoczyna się na szczytce wirtualnego przadra i rośnie w dół w kierunku adresu zerowego
 - jeśli stos rozrośnie się poniżej dolnej granicy tego segmentu, wystąpi błąd
 - programy same wprost nie zarządzają rozmiarem segmentu stosu



Motywacja

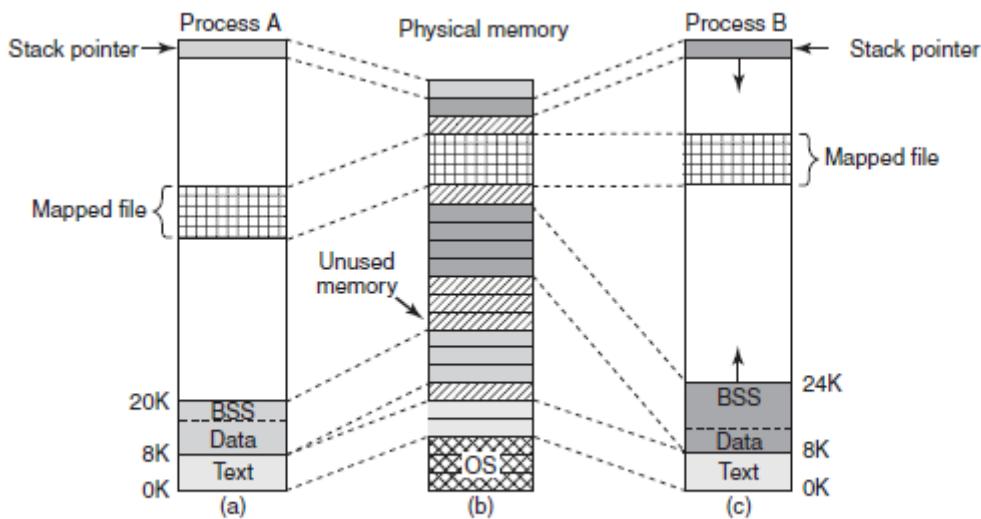
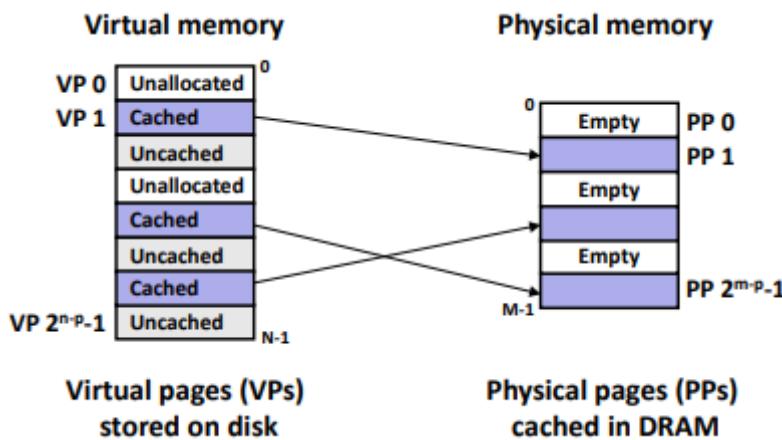
Wymiana pamięci

- poważne współczesne aplikacje użytkowe, takie jak Photoshop, często zużywają dużo pamięci na samo uruchomienie i przetwarzanie danych. W konsekwencji utrzymywanie wszystkich procesów w pamięci przez cały czas wymaga olbrzymich ilości pamięci i nie może być zrealizowane, jeśli rozmiar pamięci na to nie pozwala. Dwa ogólne rozwiązania problemu przeładowania pamięci:
 - wymiana (ang. swapping) - polega na załadowaniu określonego procesu w całości, uruchomieniu go przez pewien czas, a następnie umieszczeniu z powrotem na dysku. Bezczyńne procesy zwykle są zapisane na dysku, zatem wtedy, gdy nie działają, w ogóle nie zajmują pamięci (ale niektóre okresowo budzą się w celu wykonania swojej pracy, a następnie przechodzą w stan uśpienia).
 - pamięć wirtualna - umożliwia programom działanie nawet wtedy, gdy są częściowo zapisane w pamięci głównej

Idea VM

Pamięć wirtualna

- idea: oszukanie procesów, że każdy ma dla siebie ciągły, dużego (więcej, niż fizycznie jest dostępne) przadra (podczas gdy fizycznie może być pofragmentowany, nieciągły, częściowo przechowywany na urządzeniach pamięci masowej)
- procesy posługują się adresami wirtualnymi i żyją w wirtualnej przestrzeni adresowej. Procesor przy każdym dostępie do pamięci wykonuje translację adresów z wirtualnych na fizyczne z użyciem jednostki zarządzania pamięcią (MMU),
- wirtualna przestrzeń adresowa jest podzielona na strony. Każda strona zawiera ciągły zakres adresów wirtualnych. Strony są mapowane na pamięć fizyczną, ale nie każda strona w wirtualnej musi znajdować się w fizycznej.
- pamięć wirtualna to ciągła tablica przechowywana na dysku, a jej zawartość jest cachowana w pamięci fizycznej (DRAM cache)



a) wirtualny przadr procesu A; b) pamięć fizyczna; c) wirtualny przadr procesu B

Stronicowanie

stronicowanie (ang. *paging*) - technika zarządzania pamięcią, polegająca na podzieleniu na kawałki równej wielkości pamięci logicznej procesów (strony) i pamięci fizycznej (ramki stron); strony pamięci logicznej mogą być w dowolny sposób rozmieszczone w ramkach pamięci fizycznej

Zasada działania MMU

- do jednostki MMU przesyłany jest 16-bitowy adres wirtualny, który jest dzielony na 4-bitowy numer strony i 12-bitowe przesunięcie
- numer strony jest wykorzystywany jako indeks do tabeli stron, która zwraca numer ramki strony odpowiadającej tej stronie wirtualnej
- bit obecna/nieobecna:
 - 0 – brak strony (ang. *page fault*). System wybiera mało używaną ramkę strony i zapisuje jej zawartość na dysk (jeśli nie została tam zapisała wcześniej). Następnie pobiera stronę, do której przed chwilą program się odwoływał, na miejsce strony przed chwilą zwolnionej. Modyfikuje mapę i wznowia przerwaną instrukcję.
 - 1 – numer ramki strony znaleziony w tabeli stron jest kopowany do 3 górnych bitów rejestru wyjściowego. 12-bitowe przesunięcie jest kopowane z wchodzącego adresu wirtualnego w postaci niezmodyfikowanej. Razem tworzą one 15-bitowy adres fizyczny.

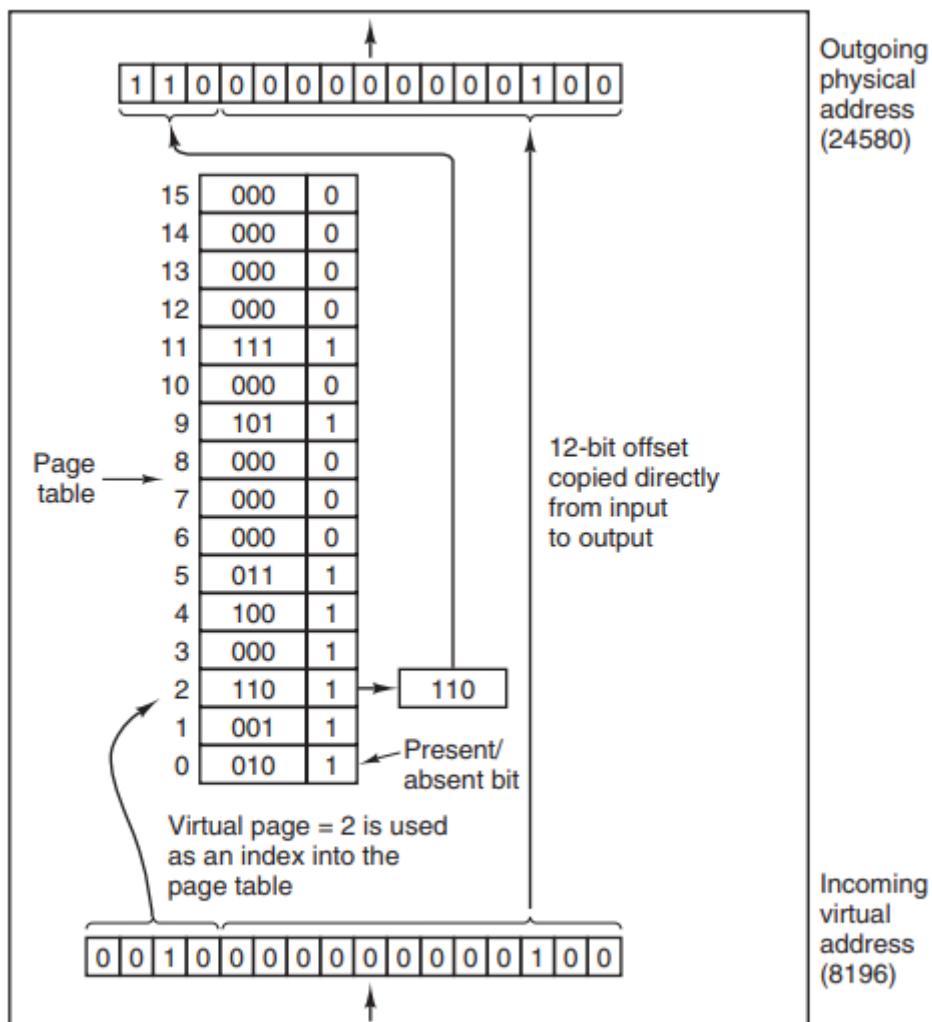
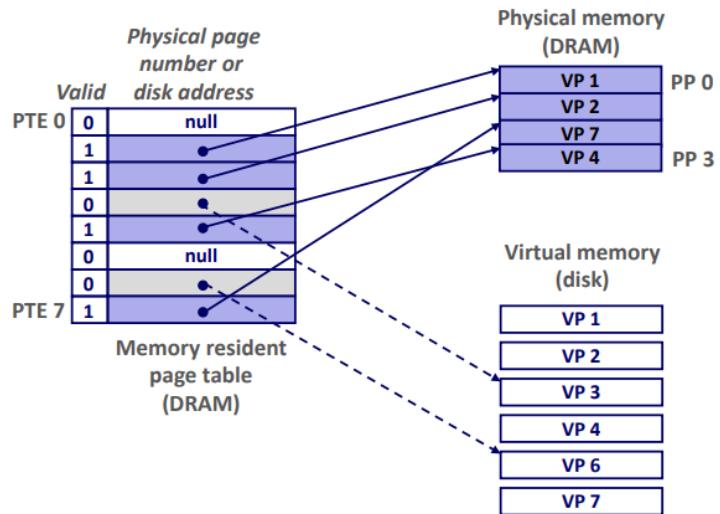


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

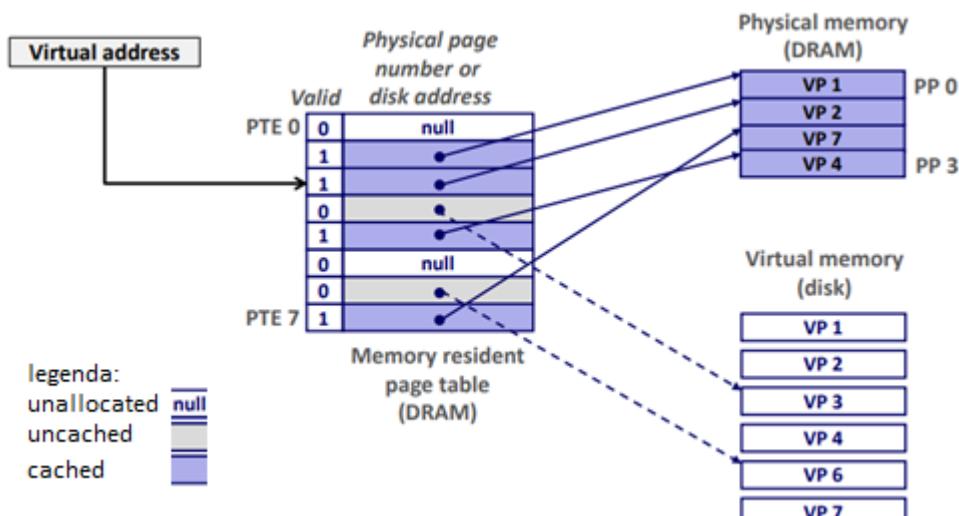
Tablica stron

tablica stron (ang. *page table*) - to tablica wpisów do tablicy stron (ang. *page table entries, PTEs*), która mapuje strony pamięci wirtualnej na strony pamięci fizycznej

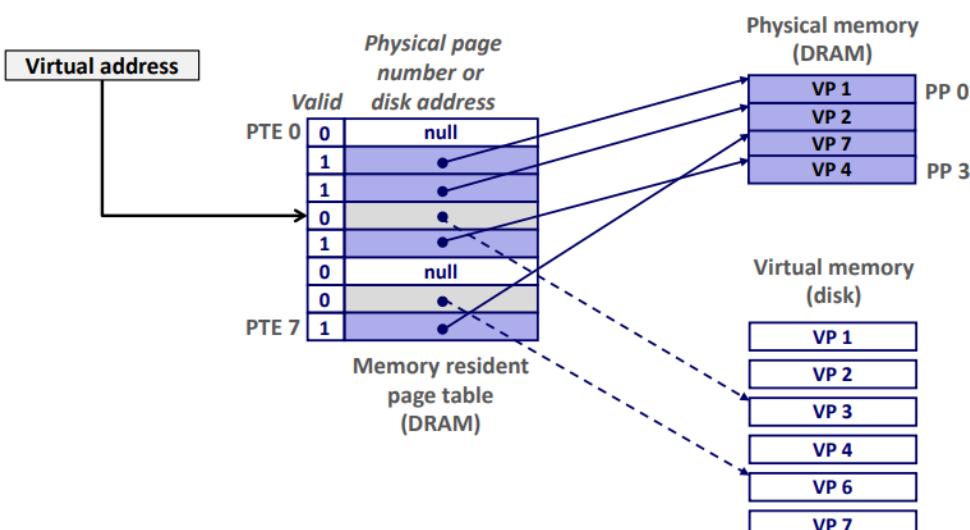


Możliwe wyniki odwołania do strony

- page hit - odwołanie do adresu wirtualnego, który znajduje się w pamięci fizycznej (DRAM cache hit)



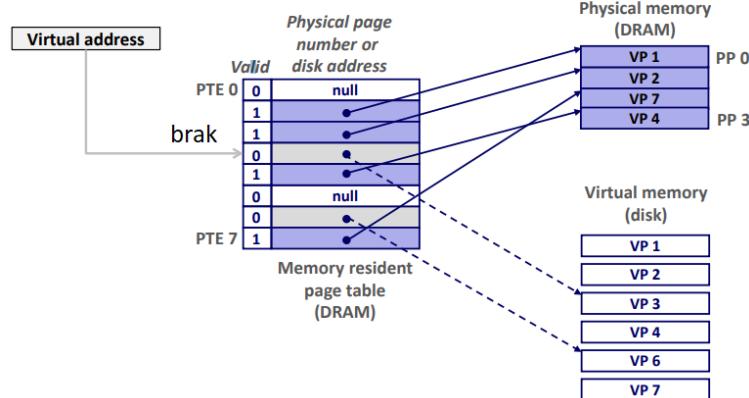
- page fault (wyjątek) – odwołanie do adresu wirtualnego, który nie znajduje się w pamięci fizycznej (DRAM cache miss)



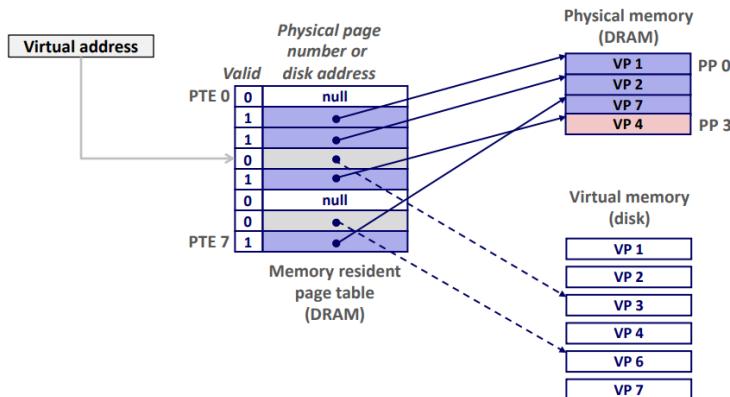
Handlowanie z page faultem

- page miss powoduje page fault (wyjątek)
- handler page faulta wybiera ofiarę do usunięcia z RAM (tutaj: PP3)

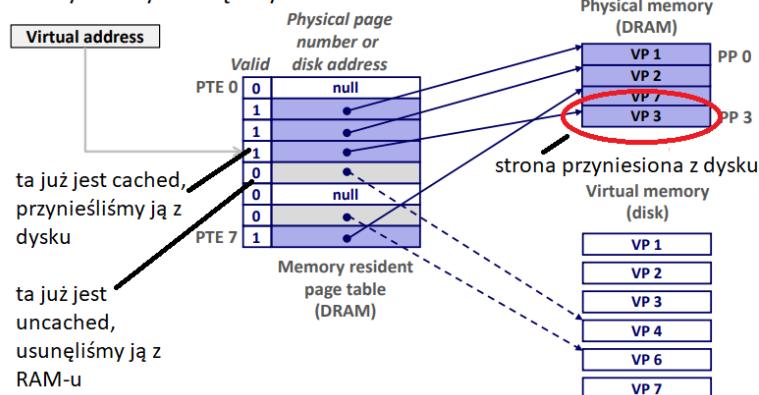
1. Ups, żądana strona nie jest skuszowana w RAM-ie.



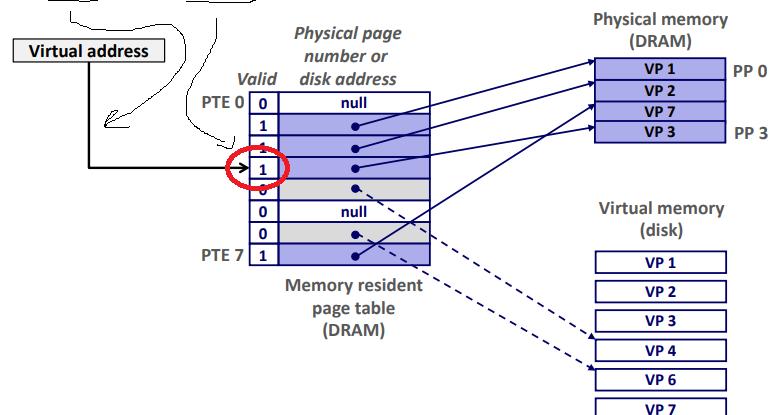
2. Wybieramy stronę-ofiarę w RAM-ie. Wywalimy PP 3, które ksesuje VP 4.



3. Przynosimy stronę z dysku do RAM-u.



4. Page fault to tylko wyjątek, więc możemy wrócić do tego, co robiliśmy. I cyk, page hit.



Struktura wpisu w tablicy stron

Wpisy w tablicy stron oprócz fizycznego numeru strony zawierają następujące bity:

- uprawnień - informuje, jakiego rodzaju dostęp jest dozwolony - czy procesor może odczytywać, zapisywać, wykonywać zawartość strony
- obecności - czy strona ma przypisaną ramkę
- trybu pracy procesora - czy procesor będąc w trybie usera lub nadzorcy może korzystać ze strony
- polityki buforowania - czy i jak procesor może przechowywać zawartość strony w pamięci podręcznej
- monitorowania dostępu
 - referenced - w użyciu - ustawiany, gdy następuje odczyt lub zapis strony. Wartość bitu pomaga w podjęciu decyzji o tym, którą stronę przerzucić na dysk w przypadku wystąpienia błędu braku strony - lepiej te, które nie są używane
 - modified - czy modyfikował tę stronę, zwany bitem zabrudzenia; bit ma sens, gdy sysopek zdecyduje się na odzyskanie ramki strony - kiedy strona jest zapisywana, sprzęt automatycznie ustawia bit modified. Jeśli nie była modyfikowana (czyli jest "czysta"), to może zostać porzucona, ponieważ na dysku znajduje się prawidłowa kopia.

Błędy strony

- gdy nie znaleziono strony w tablicy stron procesu, dochodzi do powstania błędu
 - drobny błąd strony (ang. *minor page fault*) - strona może fizycznie być przechowywana w pamięci, ale nie ma jej w tablicy stron procesu. Strona mogła być np. załadowana do pamięci z dysku przez inny proces. W tym przypadku nie ma potrzeby, by sięgać do dysku ponownie. Wystarczy odpowiednio zmapować stronę w tablicy stron.
 - poważny błąd strony (ang. *major page fault*) - gdy zachodzi konieczność sprowadzenia strony z dysku
 - błąd segmentacji (ang. *segmentation fault*) - w programie nastąpiła próba dostępu do nieprawidłowego adresu i nie ma potrzeby dodawania mapowania do TLB - sysopek zabija taki proces

Wydajność pamięci wirtualnej

Pamięć wirtualna brzmi jak coś niekoniecznie wydajnego, ale to działa, ponieważ mamy lokalność. Programy z dobrą temporal locality będą miały mniejsze zbiory robocze. Jeśli rozmiar zbioru roboczego < rozmiar RAM-u, to wydajność będzie wysoka. Jeśli rozmiar zbioru roboczego > rozmiar RAM-u, to następuje szamotanie.

- zbiór roboczy (ang. *working set*) - to zbiór stron, których proces potrzebował w chwili t do działania w ciągu Δ ostatnich tyknięć wirtualnego zegara. Podzbiór rezydentnego w delcie czasu.
- zbiór rezydentny (ang. *resident set*) - to zbiór wszystkich stron procesu rezydujących w pamięci operacyjnej w chwili t wirtualnego zegara procesu. Sysopek zarządza zbiorem rezydentnym i stara się przybliżać nim zbiór roboczy. (Zbiór rezydentny to to co proces trzyma w RAM-ie, wszystkie strony które sobie zadeklarowały, jakby zamrozić całkowicie proces, i spojrzeć na cały RAM jaki on używa, to to jest zbiór rezydentny)
- szamotanie (ang. *trashing*) - gdy proces ma mniej ramek niż liczba aktywnie używanych stron. To stan procesu, w którym spędza on więcej czasu na oczekiwaniu na brakujące strony pamięci, niż na faktycznym wykonywaniu obliczeń, co znacząco spowalnia jego działanie.

Zadanie 11 (5). Zdefiniuj pojęcie zbioru roboczego. Czym różni się on od zbioru rezydentnego?

Zbiór roboczy i rezydentny mogą się odwołać zarówno do pamięci podręcznej, jak i pamięci wirtualnej. Rozważam drugą opcję.
ZBIÓR ROBOCZY - zbiór stron z których korzysta proces w danym przediale czasu. ZBIÓR REZYDENTNY - zbiór stron procesu, które znajdują się w pamięci operacyjnej w danym przediale czasu.
Zbiór rezydentny apokrymuje zbiór roboczy i musi być wieleny od roboczego, żeby program działał efektywnie - inaczej daria błędów stron i potencjalne szamotanie.

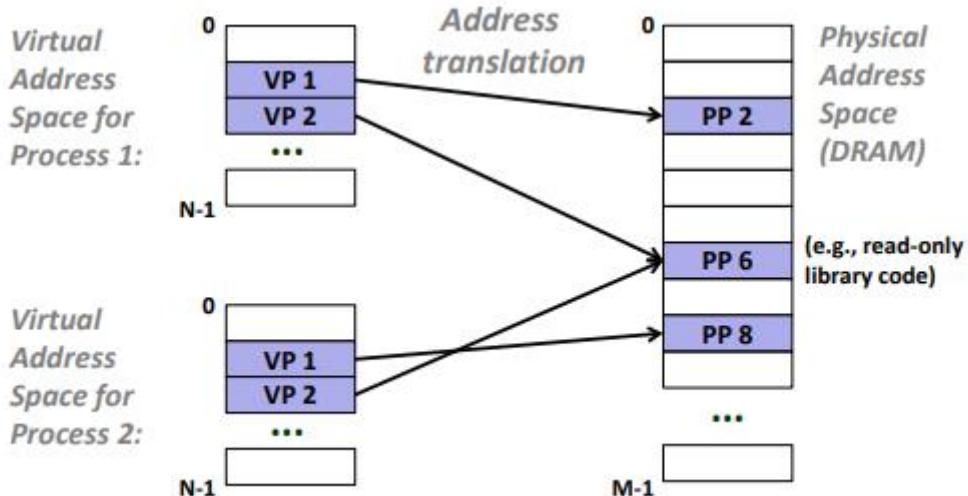
VM as a Tool for Memory Management

■ Simplifying memory allocation

- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times

■ Sharing code and data among processes

- Map virtual pages to the same physical page (here: PP 6)



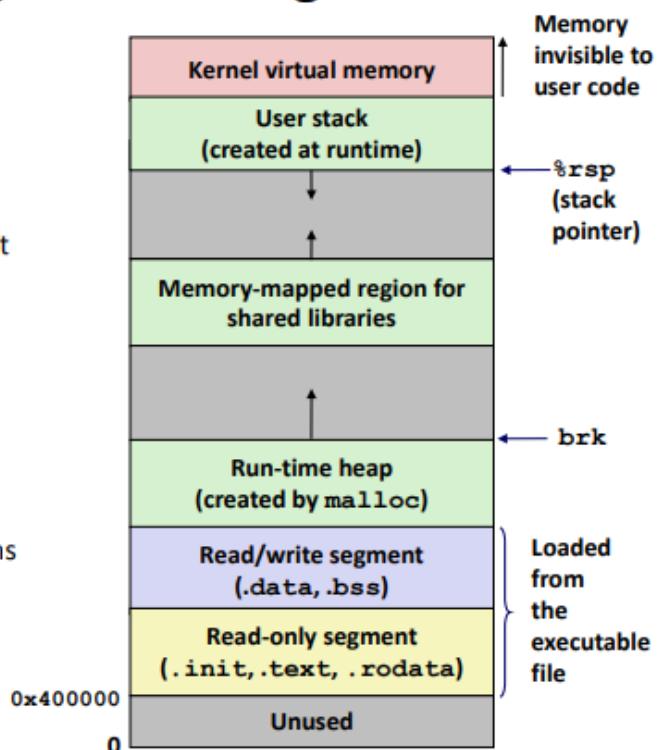
Simplifying Linking and Loading

■ Linking

- Each program has similar virtual address space
- Code, data, and heap always start at the same addresses.

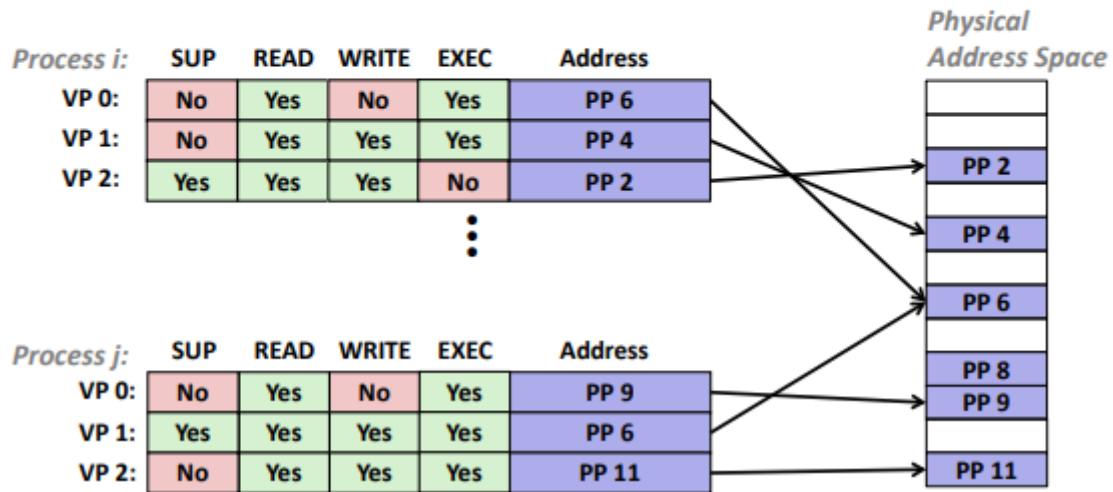
■ Loading

- `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



VM as a Tool for Memory Protection

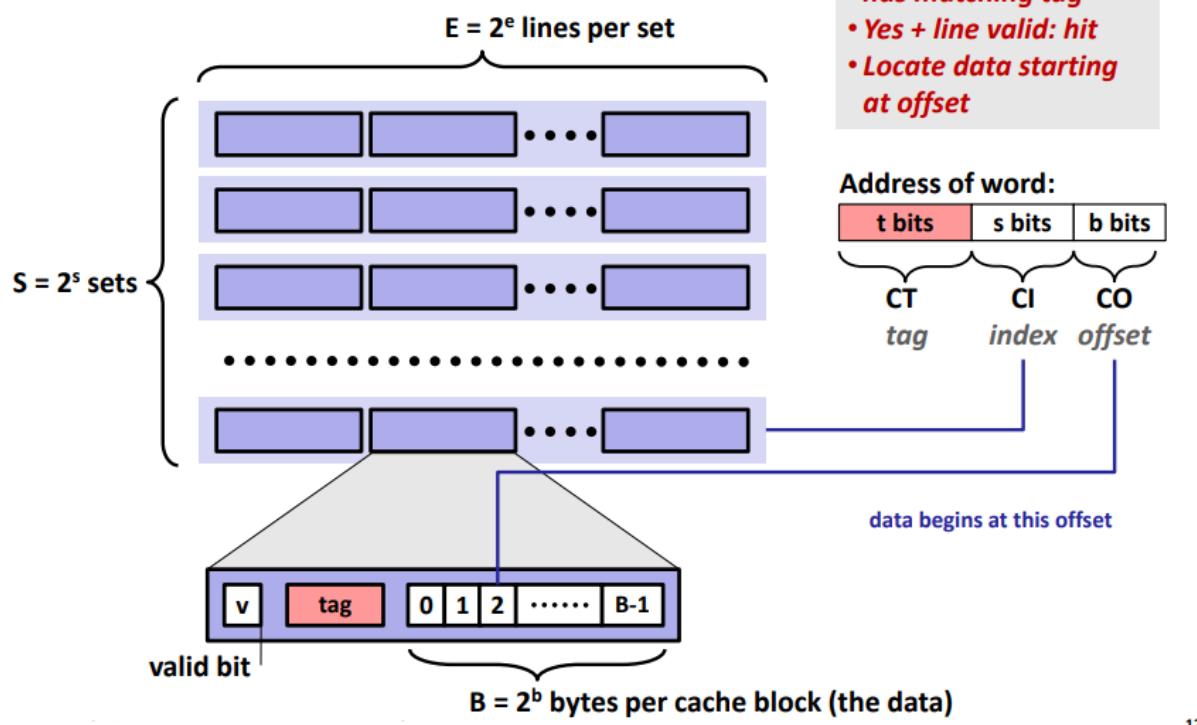
- Extend PTEs with permission bits
- MMU checks these bits on each access



Przyspieszenie stronicowania

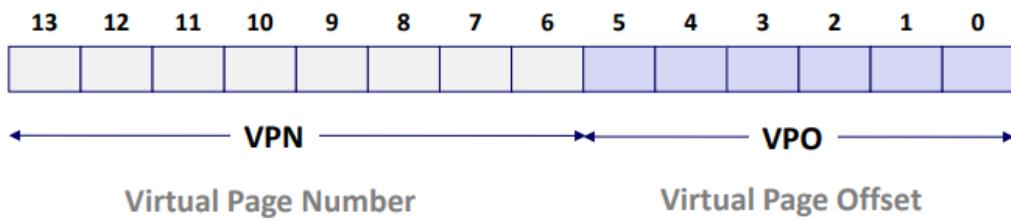
Summary słowniczek

Set Associative Cache: Read

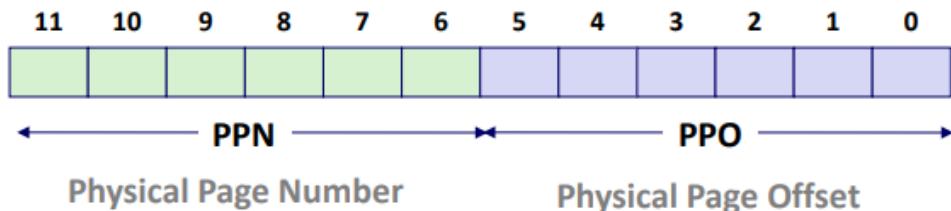


- 12

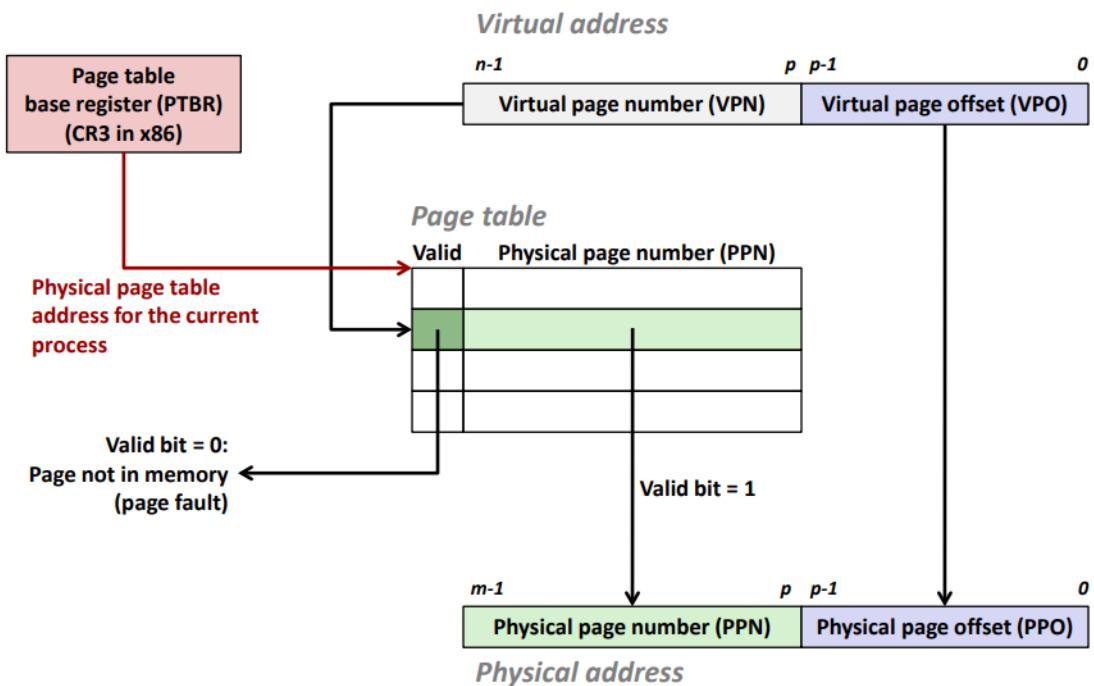
- komponenty wirtualnego adresu (VA):
 - TLBI – TLB index (o TLB za chwilę)
 - TLBT – TLB tag
 - VPO – virtual page offset
 - VPN – virtual page numer



- komponenty fizycznego adresu (PA):
 - PPO – physical page offset (= VPO)
 - PPN – physical page numer
 - CO – offset w linii cache
 - CI – cache index
 - CT – cache tag

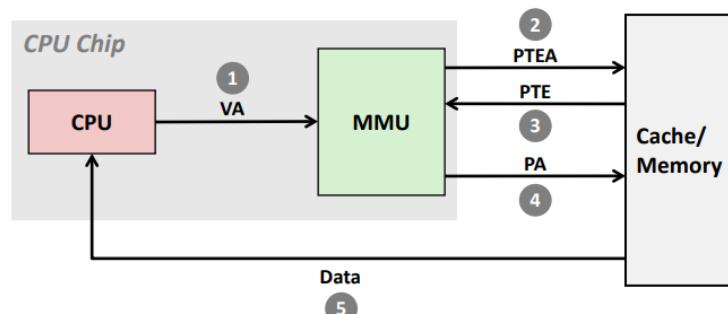


Translacja adresów z tablicą stron



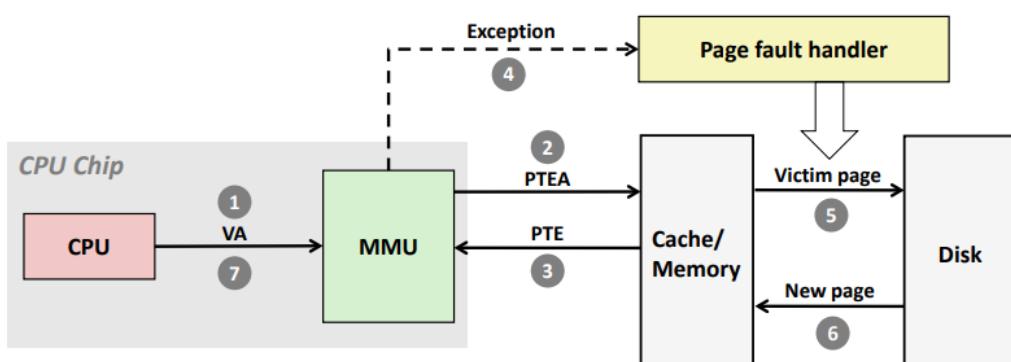
Translacja adresów: page hit

1. Procesor wysyła adres wirtualny do MMU
- 2-3. MMU pobiera PTE z tablicy stron w pamięci
4. MMU wysyła adres fizyczny do pamięci
5. Pamięć wysyła żądane słowo pod obliczonym fizycznym adresem do procesora



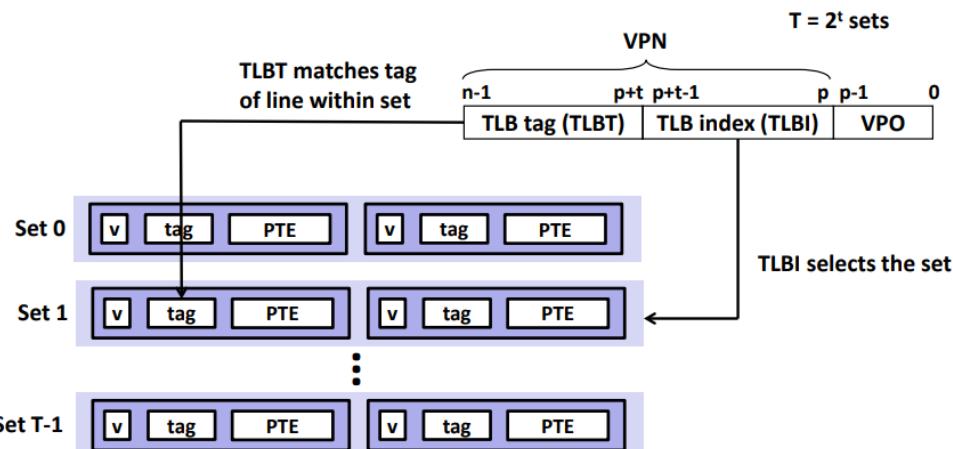
Translacja adresów: page fault

1. Procesor wysyła adres wirtualny do MMU
- 2-3. MMU pobiera PTE z tablicy stron w pamięci
4. Valid bit = 0, MMU triggeruje page fault exception
5. Handler znajduje stronę-ofiarę, jeśli jest „dirty”, to przesuwa ją na dysk
6. Handler pobiera stronę z dysku i update’uje PTE w pamięci
7. Handler wraca do procesu, restart instrukcji, która ostatnio wywołała page fault

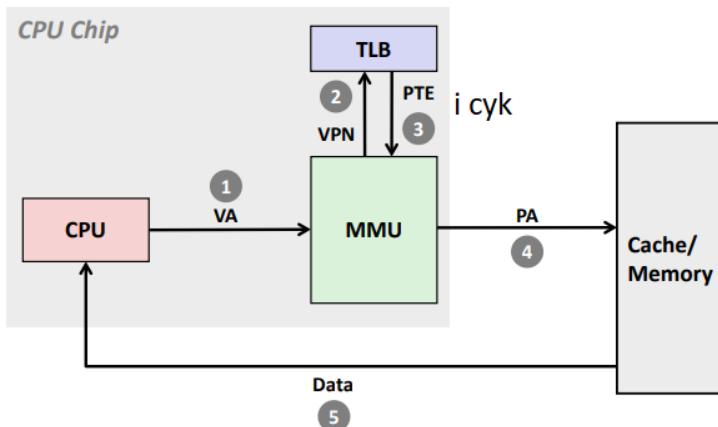


TLB

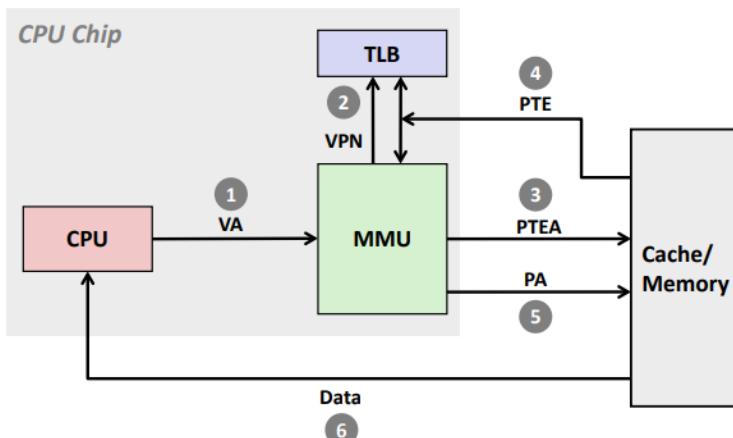
- obserwacja: większość programów zazwyczaj wykonuje wiele odwołań do małej liczby stron. Tylko niewielka część pozycji w tabeli stron jest często czytana.
- PTE-sy są cachowane w L1 jak każde inne słowo pamięci
 - mogą być ofiarą i zostać usunięte przez inne odwołania
 - mały, ale wciąż delay dostępu do L1
- rozwiązanie: TLB (ang. *Translation Lookaside Buffer*) – mały cache w MMU, który mapuje wirtualne adresy na fizyczne w tyci (ale kompletnej, jeśli chodzi o informacje) tablicy stron
- zasada działania TLB: po przesłaniu adresu wirtualnego do MMU, aby dokonać translacji, najpierw sprawdzamy, czy w VPN jest w TLB



- TLB hit – ramka strony pobierana jest bezpośrednio z TLB



- TLB miss – wyszukiwanie w tabeli; usuwa się jedną z pozycji w TLB i zastępuje odczytaną przed chwilą pozycją z tabeli stron
 - chybienie miękkie – nie ma w TLB, ale jest w pamięci; nie trzeba robić IO, tylko TLB update
 - chybienie twarde – nie ma w TLB, nie ma w pamięci; trzeba wziąć z dysku



Zadanie 6 (10). TLB jest zorganizowany jako czterodrożna pamięć sekcyjno-skojarzeniowa o 4 zbiorach. Wirtualna przestrzeń adresowa ma 2^{14} bajtów, a fizyczna 2^{12} . Rozmiar strony to 64 słowa. Polityka wymiany wpisów to *least recently used* – im wyższy numer znacznika LRU tym wpis jest starszy. Poniżej podano pierwotny stan TLB i 16 pierwszych wpisów tablicy stron. **Dane są w zapisie szesnastkowym!**

SET	TAG	PPN	LRU									
0	07	02	2	09	0D	1	00	28	0	-	-	3
1	03	2D	0	-	-	3	-	-	3	-	-	3
2	-	-	3	-	-	3	-	-	3	-	-	3
3	1A	34	0	-	-	3	-	-	-	-	-	3

VPN	PPN	VPN	PPN
0	28	8	13
1	-	9	17
2	33	A	09
3	02	B	-
4	31	C	19
5	16	D	2D
6	-	E	11
7	-	F	0D

Przetłumacz adresy wirtualne podane w poniższej tabeli. Dla każdego adresu wskaż czy chybili w TLB lub wygenerował błąd strony. Podaj też ostateczny stan TLB po przetłumaczeniu wszystkich adresów. **Udzielając odpowiedzi należy użyć zapisu szesnastkowego!** Pierwszy adres został już przetłumaczony, a modyfikacja stanu TLB została uwzględniona w tabelce wyżej.

adres wirtualny	adres fizyczny	indeks TLB	znacznik TLB	chybienie w TLB	błąd strony
0024	a24	0	00	TAK	NIE
0326	666	0	03	TAK	NIE
1ace	80e	3	1a	NIE	NIE
0216	4d6	0	02	TAK	NIE
018e	?	2	01	TAK	TAK
032a	66a	0	03	NIE	NIE

SET	TAG	PPN	LRU									
0	02	13	1	09	0D	3	00	28	2	03	19	0
1	03	20	0	-	-	3	-	-	3	-	-	3
2	-	-	3	-	-	3	-	-	3	-	-	3
3	1A	34	0	-	-	3	-	-	-	-	-	3

Zadanie 15. Stronicowanie, TLB i jednostka zarządzania pamięcią (MMU).

- MMU procesorów x86-64 może modyfikować zawartość tablicy stron bez udziału systemu operacyjnego.
- Dla każdej strony można zadać czy jej zawartość ma być przechowywalna w pamięci podręcznej.
- Stronicowanie umożliwia zarówno izolację jak i współdzielenie pamięci między procesami.
- Pamięć TLB może być indeksowana i tagowana wyłącznie adresami wirtualnymi.

Zadanie 10 (6). Jakie zadanie pełni TLB w procesie translacji adresów? Kiedy procesor modyfikuje zawartość wpisów w TLB? W jakich przypadkach system operacyjny zmienia zawartość TLB?

TLB jest pamięcią podręczną dla tablicy stron. Przechowuje niedużą (z reguły 64 dla TLB L1) liczbę wpisów tablicy stron. Tak długo jak dostęp do pamięci są w zasięgu **zasiegu TLB** to procesor nie musi czytać tablicy stron z DRAM'u. Translacja adresów musi być szybka, bo procesor wykonuje ją przy każdym dostępie do pamięci.

Procesor modyfikuje zawartość wpisów stron w TLB, jeśli program robił dostęp do strony – ustawia bit «referenced», gdy strona została odczytana, a «modified» gdy zapisana. System operacyjny używa tych bitów by analizować zbiór rezydentnych programów.

Jądro systemu operacyjnego usuwa wpisy z TLB, gdy przełącza przestrzeń adresową, jeśli procesor nie dysponuje identyfikatorami przestrzeni adresowych ASID. Dodatkowo w momencie zmiany uprawnień dostępu do strony, lub usuwania mapowania.

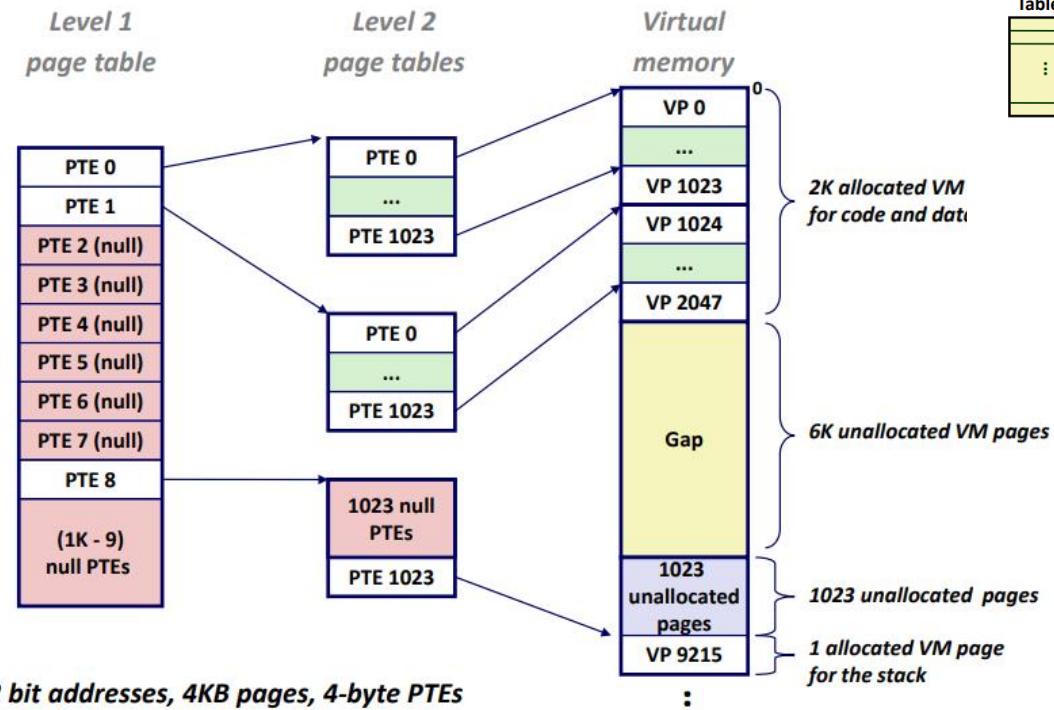
Zadanie 13 (4). Translacja adresów i TLB.

- Hierarchiczna tablica stron umożliwia oszczędniejsze opisywanie przestrzeni adresowej niż płaska tablica stron.
- TLB przechowuje strony pamięci wirtualnej.
- Procesor może modyfikować bity dodatkowe we wpisach tablicy stron bez udziału systemu operacyjnego.
- Jeśli strona nie ma przypisanej żadnej ramki, to dostęp do niej generuje błąd strony.

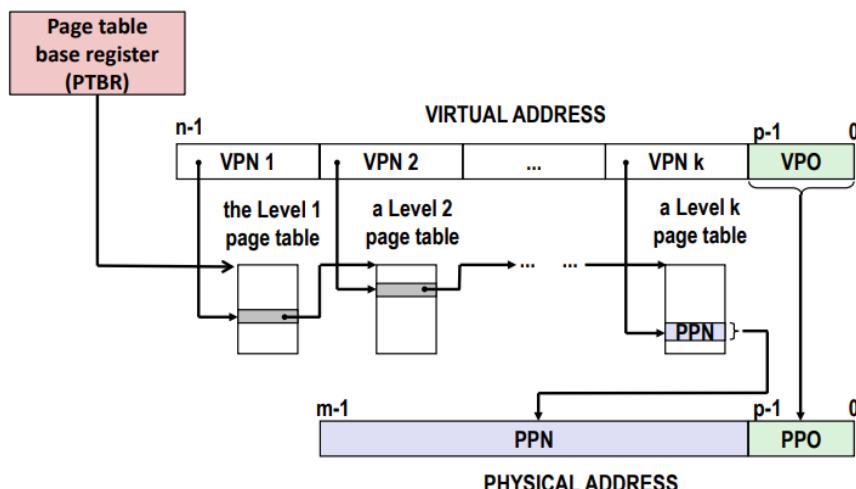
Wielopoziomowe tablice stron

Motywacja

- duża przestrzeń adresowa powoduje, że tablica stron zajmuje dużo miejsca
- rozwiązanie: wielopoziomowa tablica stron, np. 2-poziomowa
 - pierwszy poziom: każdy PTE wskazuje na tablicę stron drugiego poziomu
 - drugi poziom: każdy PTE wskazuje na stronę



Translacja z k-poziomową tablicą stron



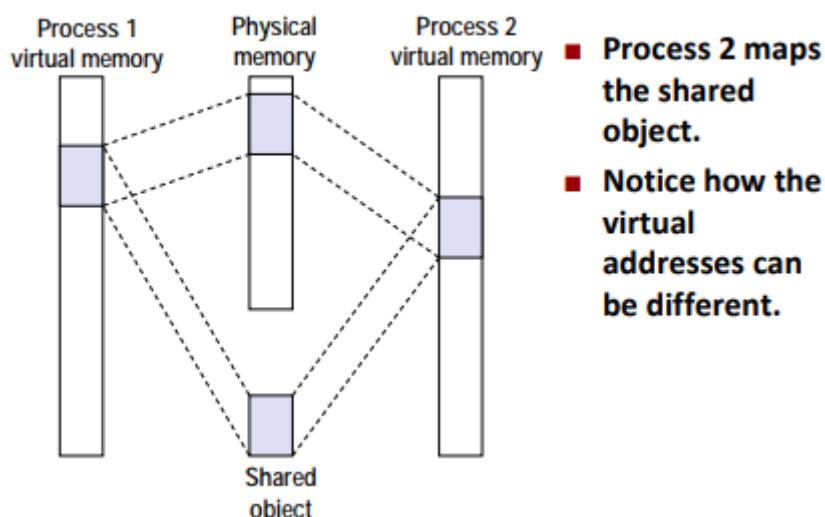
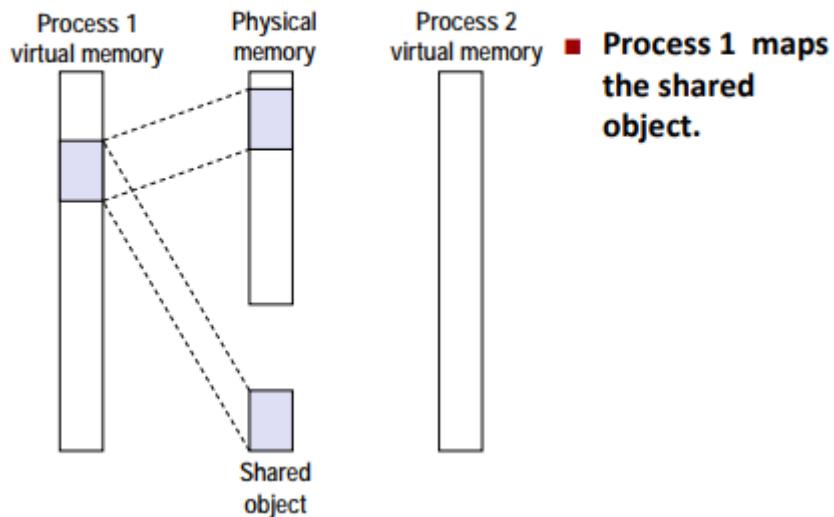
Zadanie 9 (5). Wytłumacz zwięzle proces translacji adresów z użyciem dwupoziomowej tablicy stron. Jakie informacje przechowują wpisy tablicy stron?

Adres虚拟ny dzielimy na części (architektura 32-bit):
 [101 101 off]. Procesor przekonuje w przypisówkach rejestrze mniej do katalogu stron (PTR). Wpis katalogu (PDT) PDE (PDT) stron (PDE) zawiera wskaźniki na fragment tablic stron. Wpis PTE zawiera numer rameki (PPN) który uprzedni i wyróżnia systemowe, t.j. cache, accessed, modified. Adres fizyczny obliczamy następująco: PA = PTR[VA.i02] [VA.i1], PPN << 12 + VA.off.

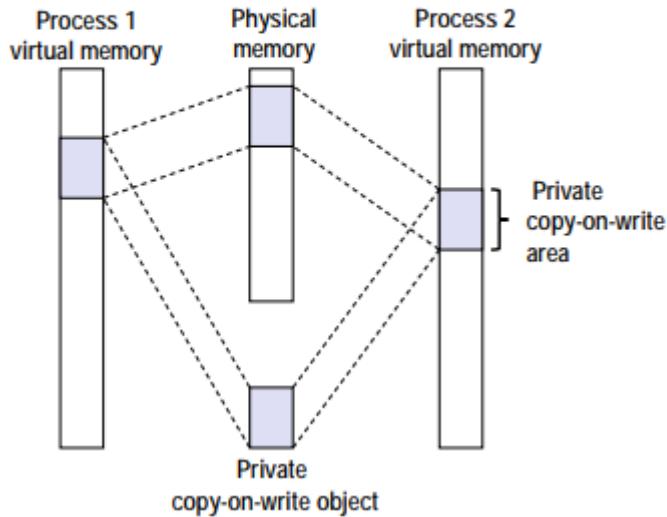
Memory Mapping

- VM areas initialized by associating them with disk objects.
 - Process is known as **memory mapping**.
- Area can be *backed* by (i.e., get its initial values from) :
 - **Regular file** on disk (e.g., an executable object file)
 - Initial page bytes come from a section of a file
 - **Anonymous file** (e.g., nothing)
 - First fault will allocate a physical page full of 0's (**demand-zero page**)
 - Once the page is written to (**dirtied**), it is like any other page
- Dirty pages are copied back and forth between memory and a special **swap file**.

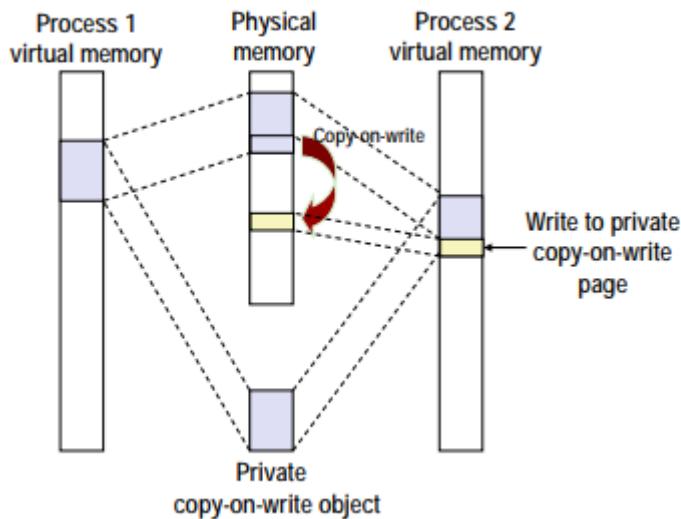
Sharing Revisited: Shared Objects



Sharing Revisited: Private Copy-on-write (COW) Objects



- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

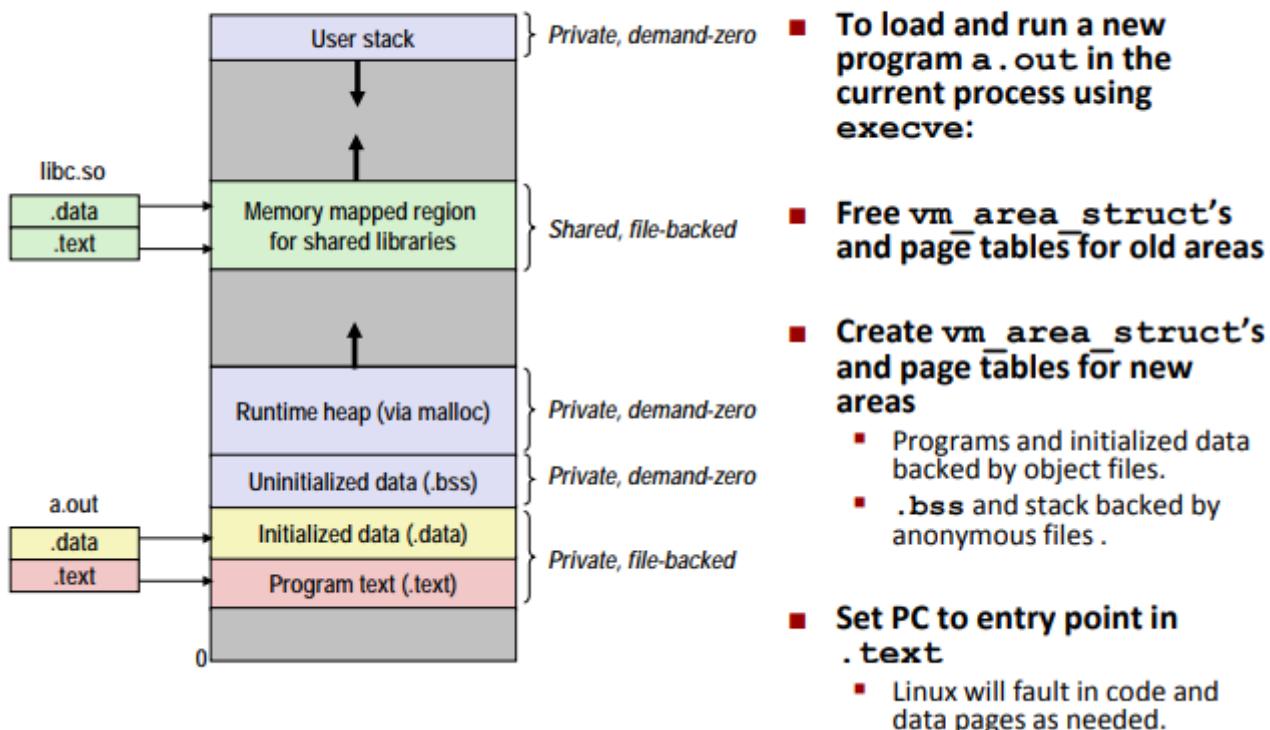


- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The fork Function Revisited

- VM and memory mapping explain how `fork` provides private address space for each process.
- To create virtual address for new new process
 - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
 - Flag each page in both processes as read-only
 - Flag each `vm_area_struct` in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new pages using COW mechanism.

The execve Function Revisited

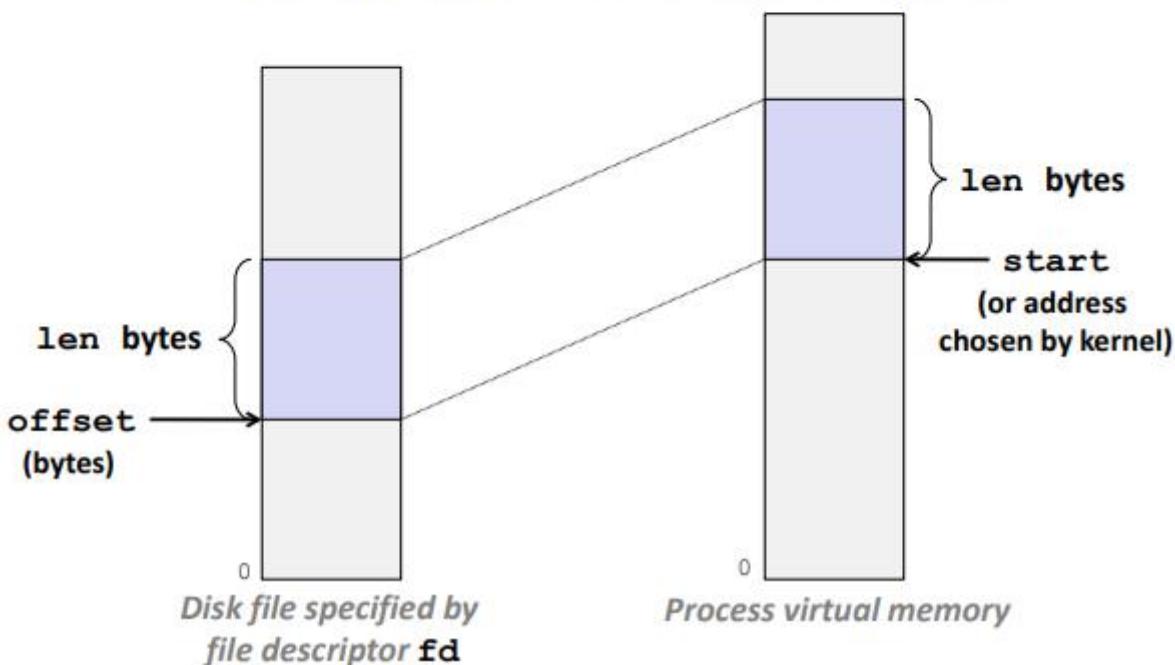


User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map **len** bytes starting at offset **offset** of the file specified by file description **fd**, preferably at address **start**
 - **start**: may be 0 for “pick an address”
 - **prot**: PROT_READ, PROT_WRITE, ...
 - **flags**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- Return a pointer to start of mapped area (may not be **start**)

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

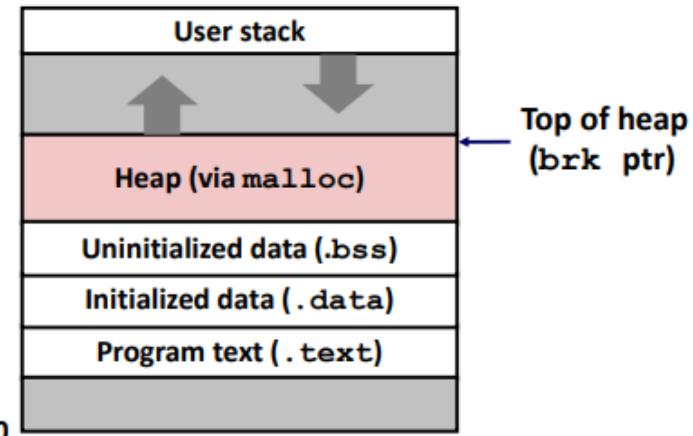
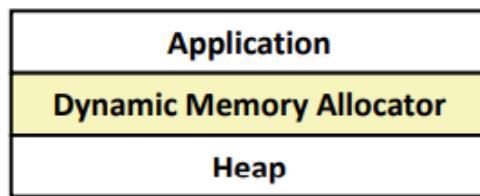


Dynamic Memory Allocation

- Programmers use **dynamic memory allocators** (such as `malloc`) to acquire VM at run time.

- For data structures whose size is only known at runtime.

- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.**



- Allocator maintains heap as collection of variable sized **blocks**, which are either **allocated** or **free**
- Types of allocators**

- Explicit allocator:** application allocates and frees space
 - E.g., `malloc` and `free` in C
- Implicit allocator:** application allocates, but does not free space
 - E.g. garbage collection in Java, ML, and Lisp

The `malloc` Package

```

#include <stdlib.h>

void *malloc(size_t size)
  ▪ Successful:
    ▪ Returns a pointer to a memory block of at least size bytes
      aligned to an 16-byte boundary (on x86-64)
    ▪ If size == 0, returns NULL
  ▪ Unsuccessful: returns NULL (0) and sets errno

void free(void *p)
  ▪ Returns the block pointed at by p to pool of available memory
  ▪ p must come from a previous call to malloc or realloc
  
```

Other functions

- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc`: Changes the size of a previously allocated block.
- `sbrk`: Used internally by allocators to grow or shrink the heap

Constraints

■ Applications

- Can issue arbitrary sequence of `malloc` and `free` requests
- `free` request must be to a `malloc`'d block

■ Allocators

- Can't control number or size of allocated blocks
- Must respond immediately to `malloc` requests
 - *i.e.*, can't reorder or buffer requests
- Must allocate blocks from free memory
 - *i.e.*, can only place allocated blocks in free memory
- Must align blocks so they satisfy all alignment requirements
 - 16-byte (x86-64) alignment on Linux boxes
- Can manipulate and modify only free memory
- Can't move the allocated blocks once they are `malloc`'d
 - *i.e.*, compaction is not allowed

Performance Goal: Throughput

■ Given some sequence of `malloc` and `free` requests:

- $R_0, R_1, \dots, R_k, \dots, R_{n-1}$

■ Goals: maximize throughput and peak memory utilization

- These goals are often conflicting

■ Throughput:

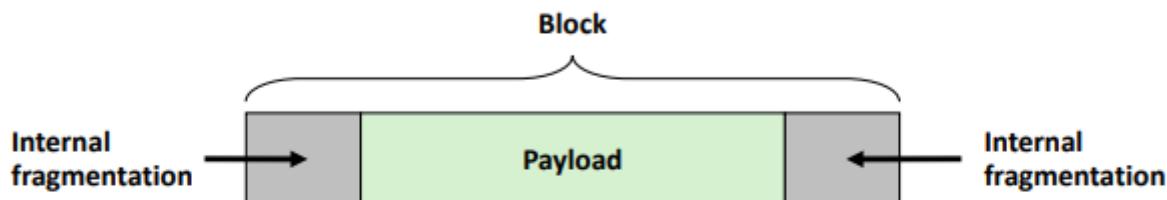
- Number of completed requests per unit time
- Example:
 - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
 - Throughput is 1,000 operations/second

Fragmentation

- Poor memory utilization caused by *fragmentation*
 - *internal* fragmentation
 - *external* fragmentation

Internal Fragmentation

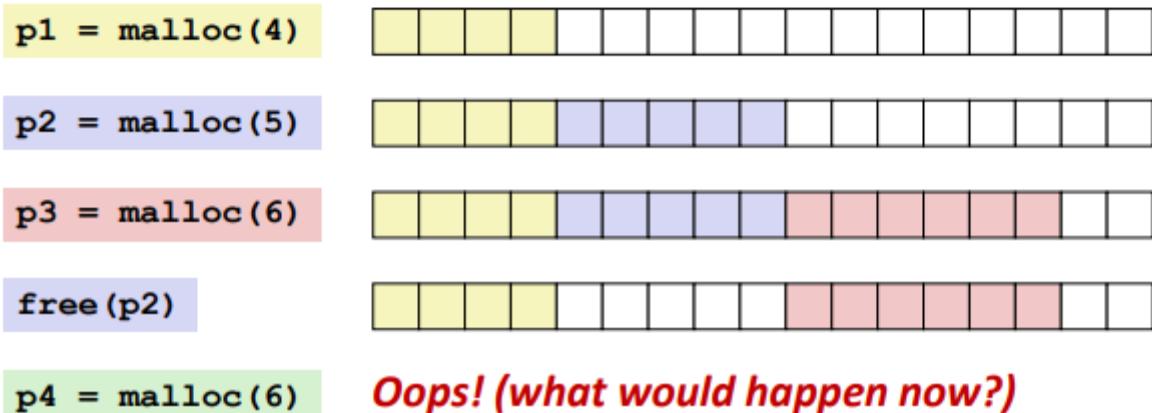
- For a given block, *internal fragmentation* occurs if payload is smaller than block size



- Caused by
 - Overhead of maintaining heap data structures
 - Padding for alignment purposes
 - Explicit policy decisions
(e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
 - Thus, easy to measure

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough



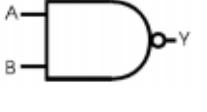
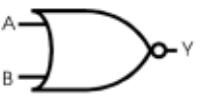
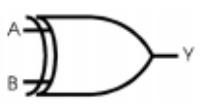
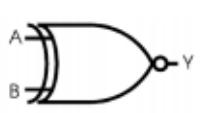
Układy cyfrowe

Podstawy

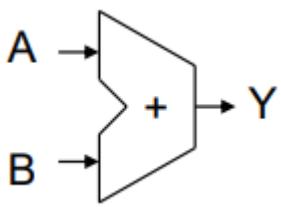
- informacje zakodowane binarnie: niskie napięcie = 0, wysokie napięcie = 1
- jeden drut na bit
- wielobitowe dane kodowane w wieloprzewodowych szynach
- dwie klasy układów cyfrowych = logicznych:
 - układy kombinacyjne – układy „bez pamięci”, stan wyjścia zależy wyłącznie od stanu wejść
 - układy sekwencyjne – układy „z pamięcią”, stan wyjścia zależy od stanu wejść układu oraz od poprzedniego stanu, zwanego stanem wewnętrznym, pamiętanego w zespole rejestrów (pamięci)

Elementy kombinacyjne:

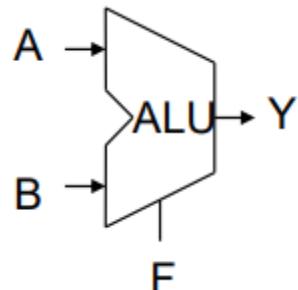
- bramki logiczne

Funkcja logiczna	Symbol logiczny	Wyrażenie algebraiczne	Tabela prawdy	
			Wejście A	Wyjście Y
AND		$A \cdot B = Y$	0	0
			0	1
			1	0
			1	1
OR		$A + B = Y$	0	0
			0	1
			1	0
			1	1
NOT		$\bar{A} = Y$	0	1
			1	0
NAND		$\overline{A \cdot B} = Y$	0	0
			0	1
			1	0
			1	1
NOR		$\overline{A + B} = Y$	0	1
			0	0
			1	0
			1	0
XOR		$A \oplus B = Y$	0	0
			0	1
			1	0
			1	1
XNOR		$\overline{A \oplus B} = Y$	0	1
			0	0
			1	0
			1	1

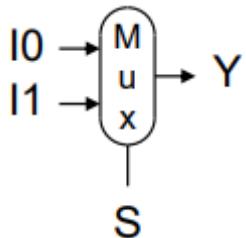
- sumator: $Y = A + B$



- Arithmetic/Logic Unit: $Y = F(A, B)$



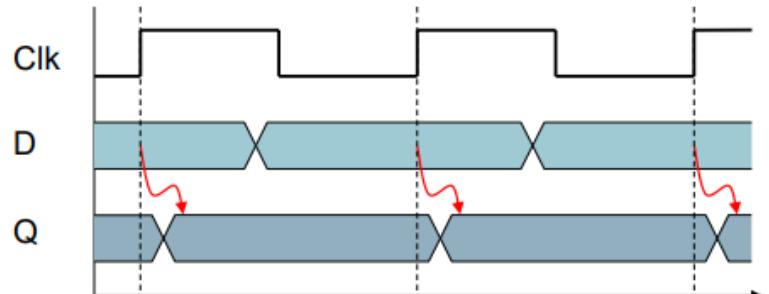
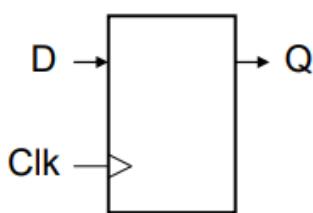
- multiplexer: $Y = S ? I_1 : I_0$



Elementy sekwencyjne:

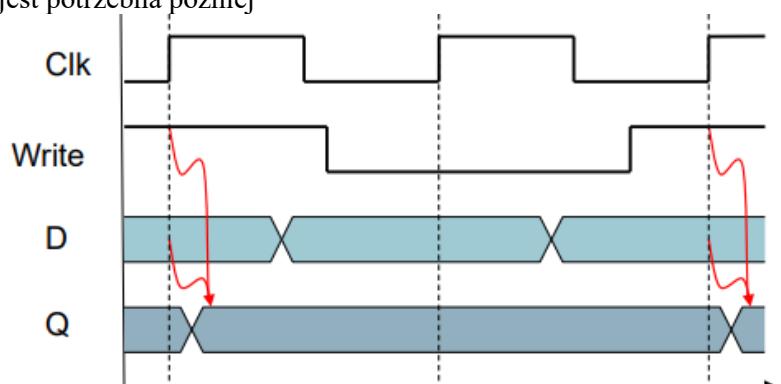
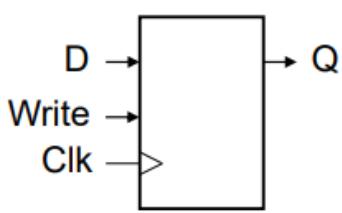
- rejestry

- zapisują dane w obwodzie
- używają sygnału zegarowego do określenia, kiedy zaktualizować zapisaną wartość
- wyzwalały zboczem: aktualizacja, gdy Clk zmienia się z 0 na 1



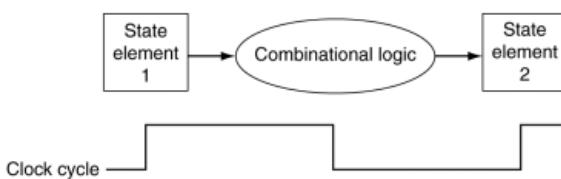
- rejestry z kontrolą zapisu

- zapis tylko na zboczu zegara, gdy write control input = 1
- używany, gdy zapisywana wartość jest potrzebna później



Clocking methodology

- logika kombinacyjna przekształca dane podczas cykli zegara
 - pomiędzy zboczami zegara
 - input z elementu sekwencyjnego, output do elementu sekwencyjnego
 - najdłuższy delay wyznacza cykl zegara



Procesor MIPS

Podstawy

- wszystkie instrukcje są 4-bajtowe
- rejesty ogólnego użytku są oznaczane znakiem \$
- wyróżniamy 3 typy instrukcji:
 - R - wszystkie dane używane przez instrukcje są w rejestrach
 - I - instrukcja operuje na stałych i wartościach rejestrów
 - J - musi być wykonany skok

Instrukcje typu R

Instrukcje

- przykłady: add, sub, or, and, div, mult, nor, xor, slt, sll, srl
- format:

OP	rd, rs, rt	//rd = rs (op) rt
◦ OP – mnemonik		
◦ rs, rt	– source, target registers	
◦ rd	– destination register	

Kodowanie instrukcji

- funct rozróżnia, jaka funkcja ALU ma się wykonać
- shamt – shift amount

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

Instrukcje typu I

Instrukcje

- przykłady: addi, beq, bne, lw, sw, ori, andi, slti
- stałe mają maks. 16 bitów
- format:

addi	OP rs, rt, IMM
lw	rt, imm(rs) //rt = MEM[rs + imm]
sw	rt, imm(rs) //MEM[rt + imm] = rs
beq	rs, rt, label

Kodowanie instrukcji

opcode	rs	rt	imm
6	5	5	16

Instrukcje typu J

Instrukcje

- przykład: j
- format: OP label

Kodowanie instrukcji

opcode	addr
6	26

Pipeline MIPS-a

Fazy instrukcji

- IF – instruction fetch, pobranie instrukcji z pamięci (adres następnej w rejestrze PC) i zapisanie instrukcji w IR (instruction register); traktuj to jako „przyszedł kurier z paczką-instrukcją, nie wiemy, co jest w środku”
- ID – instruction decode, dekoduje instrukcję w IR – tutaj się dowiadujemy, co to za instrukcja; traktuj to jako „unboxing paczki, którą przyniósł kurier”
- EX – execute – tutaj działa ALU i wykonują się wszystkie obliczenia arytmetyczne
- MEM – memory access – wykonujemy wszystkie dostępy do pamięci: w tej fazie „działa” instrukcja lw i sw, reszta nie
- WB – write back – jeśli instrukcja ma rejestr wynikowy, to w tej fazie wynik operacji jest zapisywany do destination register

Skoki

Mamy coś takiego jak predykcja skoków. Żeby nie marnować czasu na obliczenie, czy faktycznie warunek skoku jest spełniony, możemy szybciej podjąć decyzję, czy skakać, czy nie. Jak się uda, to fajnie, jak nie, to najwyżej się wycofamy.

Weźmy warunek wyjścia z pętli, która ma sto iteracji. Przez pierwsze kilka iteracji predyktor może się mylić, ale z czasem zauważymy, że ciągle zostajemy w pętli, więc zmienią taktykę na „nie wykonuj skoku” i nie będziemy wyskakiwać z pętli. Przez następne (sto – kilka) razy będzie przewidywał poprawnie, programik będzie działał szybko, pomyli się dopiero przy ostatniej iteracji, gdy faktycznie trzeba będzie wyskoczyć z pętli.

Skoki: bne, beq, j (bezwarunkowy), jr (jump na adres zapisany w rejestrze)

Niewykonane skoki

Na egzaminie powinno być podane, jaką mamy strategię, czy „zawsze wykonaj skok”, czy „nigdy nie skacz”, a skoki w kodzie powinny zostać opatrzone informacją, czy się wykonały, czy nie.

Przykład

kod 1

```
j      L2
L1:  lw    $t0, ($a0)
      beq   $t0, $a2, L3
      sw    $t0, 4($a0)
      addi  $a0, $a0, 32
L2:  bne   $a0, $a1, L1
L3:  jr    $ra
```

kod 2

```
lw    $3, 0($5)
srl   $8, $8, 1
beq   $2, $3, L1      # skok nie wykonał się
lw    $3, 0($6)
addi  $3, $3, -16
L1:  bne   $2, $3, L2      # skok wykonał się
      subu $2, $2, $8
L2:  sw    $2, -4($7)
```

Możliwe scenariusze

- (kod 1) nigdy nie skacz, instrukcja j - skoro mamy nigdy nie skakać, to ignorujemy J i ładujemy następną w kolejności instrukcję, lw. W fazie ID robimy unboxing instrukcji i okazuje się, że to zła instrukcja. Trzeba ją całą wycofać. Ładujemy instrukcję, która powinna się wykonać po skoku.

j L2	IF	ID	EX	MEM	WB			
lw \$t0,(\$a0)		IF	X kurde pszypau, miałem jednak skoczyć	X	X	X		
bne \$a0,\$a1,L1			IF	ID skoczyłem	EX	MEM	WB	

- (kod 2) nigdy nie skacz, skok ma się nie wykonać – i cyk ładniutko pobiera następną kolejną instrukcję, ładnie przewidział

beq \$2, \$3, L1		IF	ID	IF	EX	MEM	WB	
lw \$3, 0(\$6)			IF	ID	EX	MEM	WB	

Hazardy

Zapamiętaj: hazard to problem, obejście to rozwiązanie.

Dokładniej: hazard to sytuacja, gdy nie możesz zacząć następnej instrukcji w następnym cyklu po bożemu, bo jest jakiś problem. Instrukcja zależy od wyniku obliczeń lub innych działań poprzedniej instrukcji, np. instrukcja (k+1)-sza korzysta z sumy, którą obliczamy w instrukcji k-tej.

Obejścia

Normalnie musielibyśmy czekać, aż cała poprzednia instrukcja się wykona (tj. jej 5 faz), ale nie trzeba. Można zrobić obejście i pobrać dane z miejsca, w którym są one już obliczone/wprowadzone.

Producenci danych

- lw ładuje dane do rejestru w MEM
- arytmetyczne produkują wynik obliczeń w EX

Konsumenci danych

- arytmetyczne potrzebują dane w EX
- skoki warunkowe podejmują decyzję o skoku w ID (czyli np. beq musi porównać, czy dwa rejesty są równe, a jeśli poprzednia instrukcja zmieniała wartość jednego z nich operacją arytmetyczną, to beq podejmie decyzję w ID, ale weźmie dane obejściem z fazy EX poprzedniej instrukcji)

Bąbelki, chmurki, stalls, wstrzymanie wykonania

- instrukcje wstrzymujemy, gdy nie mogą się wykonać, bo np. potrzeba jeszcze jednego taktu, żeby wykonać obejście

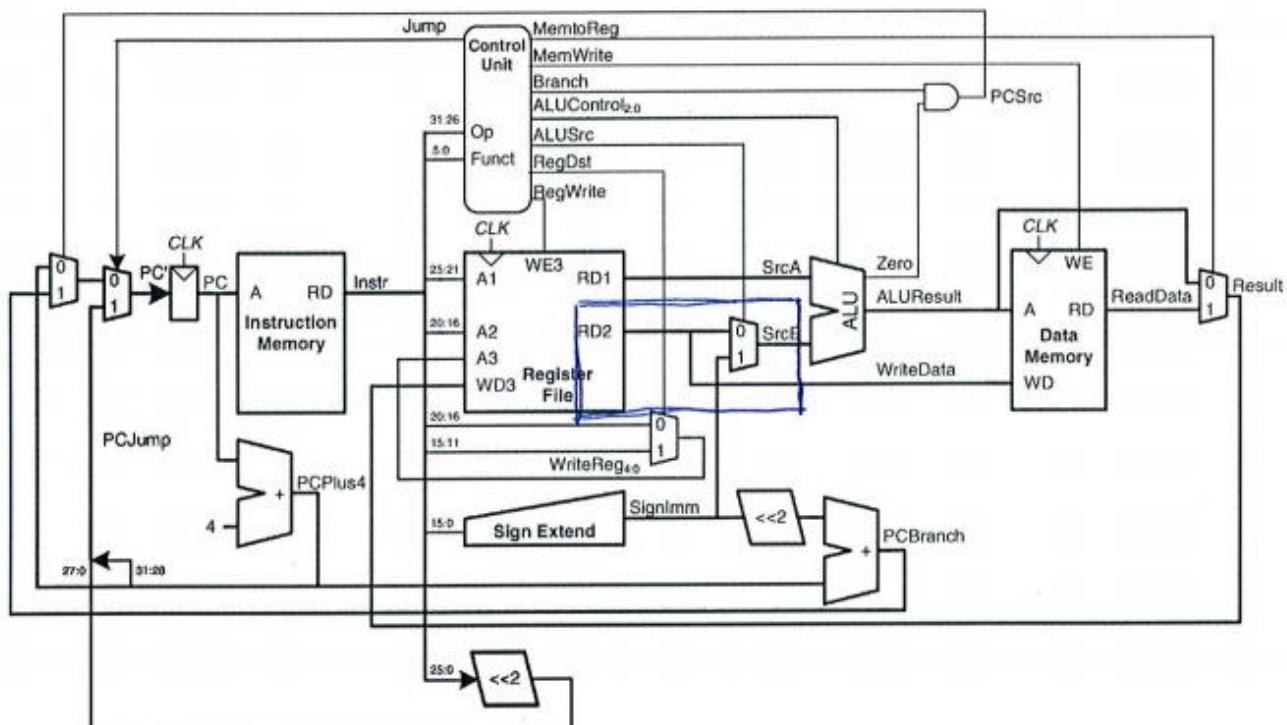
lw \$3, 0(\$6)		IF	IF	ID	EX	MEM tap	WB	
addi \$3, \$3, -16			IF	ID		EX gib dane	EX ok dzięki	MEM WB

- IFwstrzymujemy aż do momentu, gdy ustawi się pod poprawnie wykonanym ID poprzedniej instrukcji

bne \$2, \$3, L2		IF	ID	ID	IF	EX	MEM	WB
subu \$2, \$2, \$8					IF czeka	IF czeka	IF ok	ID EX MEM WB

Zadanie 7 (8). Na poniższym schemacie widnieje jednocyklowa implementacja procesora MIPS. Zaproponuj kodowanie instrukcji, nowe sygnały kontrolne i modyfikacje schematu niezbędne do obsługi dodatkowej instrukcji. Podaj stan wszystkich sygnałów sterujących. Modyfikowanie ALU jest **zabronione!**

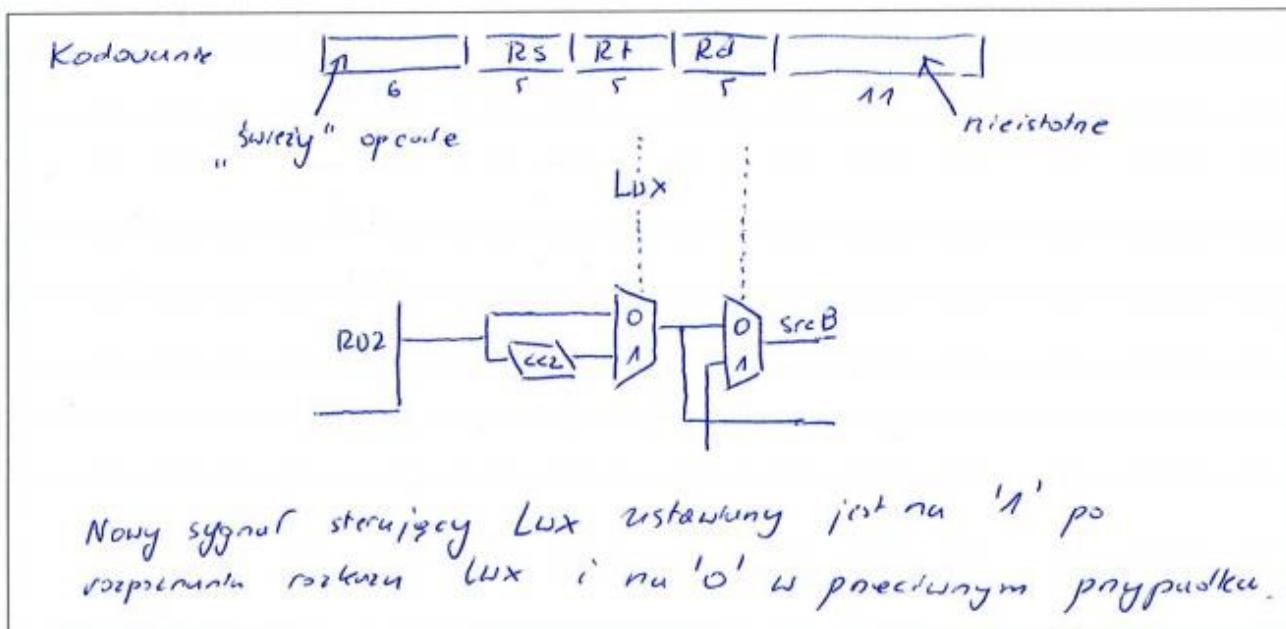
UWAGA: Na schemacie należy jedynie zaznaczyć modyfikowany fragment procesora i przerysować go ze zmianami do kratki!



mnemonik	typ	semantyka
lwx \$Rd,\$Rs[\$Rt]	R	Reg[Rd] := Mem[Reg[Rs] + Reg[Rt] * 4]

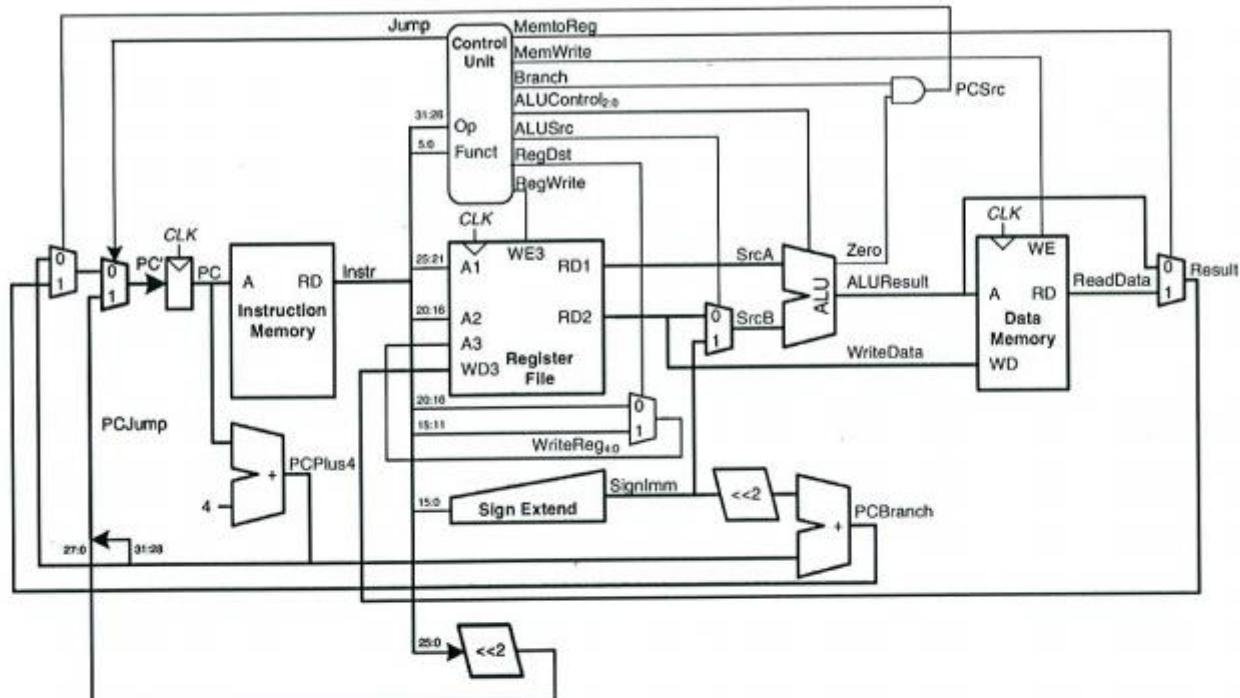
	MemToReg	MemWrite	Branch	ALUControl _{2:0}	ALUSrc	RegDst	RegWrite	Jump
lwx	1	0	0	sygnur '+'	0	1	1	0

Lwx
1



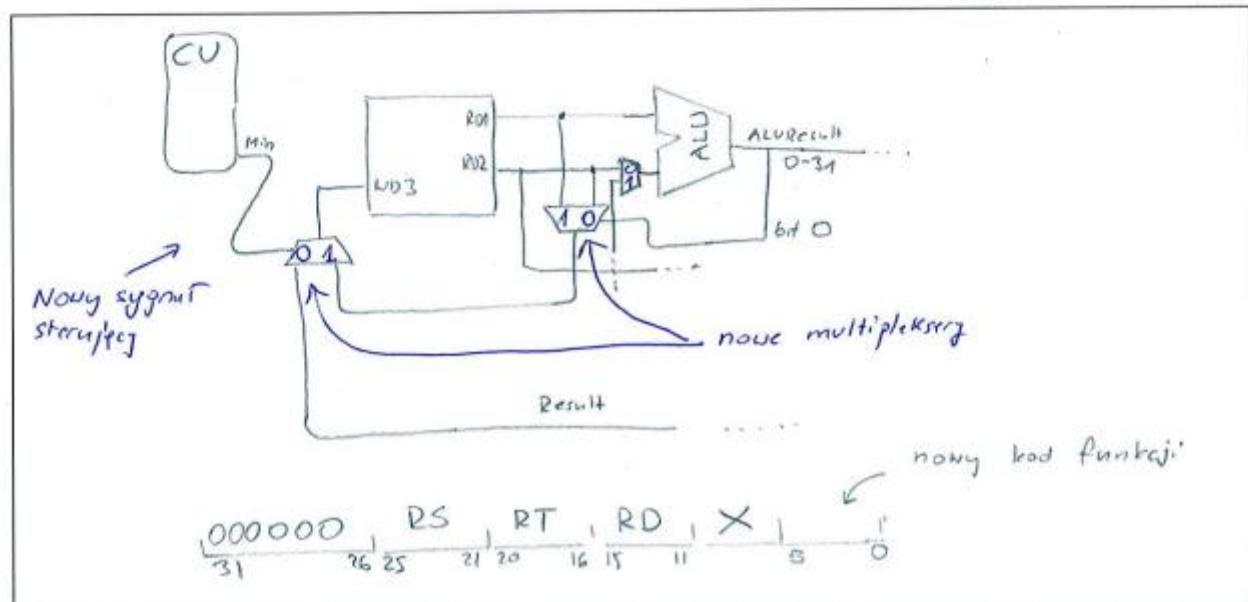
Zadanie 7 (8). Na poniższym schemacie widnieje jednocyklowa implementacja procesora MIPS. Zaproponuj kodowanie instrukcji, nowe sygnały kontrolne i minimalne modyfikacje schematu niezbędne do obsługi dodatkowej instrukcji. Podaj stan wszystkich sygnałów sterujących. Modyfikowanie ALU jest zabronione!

UWAGA: Na schemacie należy jedynie zaznaczyć modyfikowany fragment procesora i przerysować go ze zmianami do kratki!

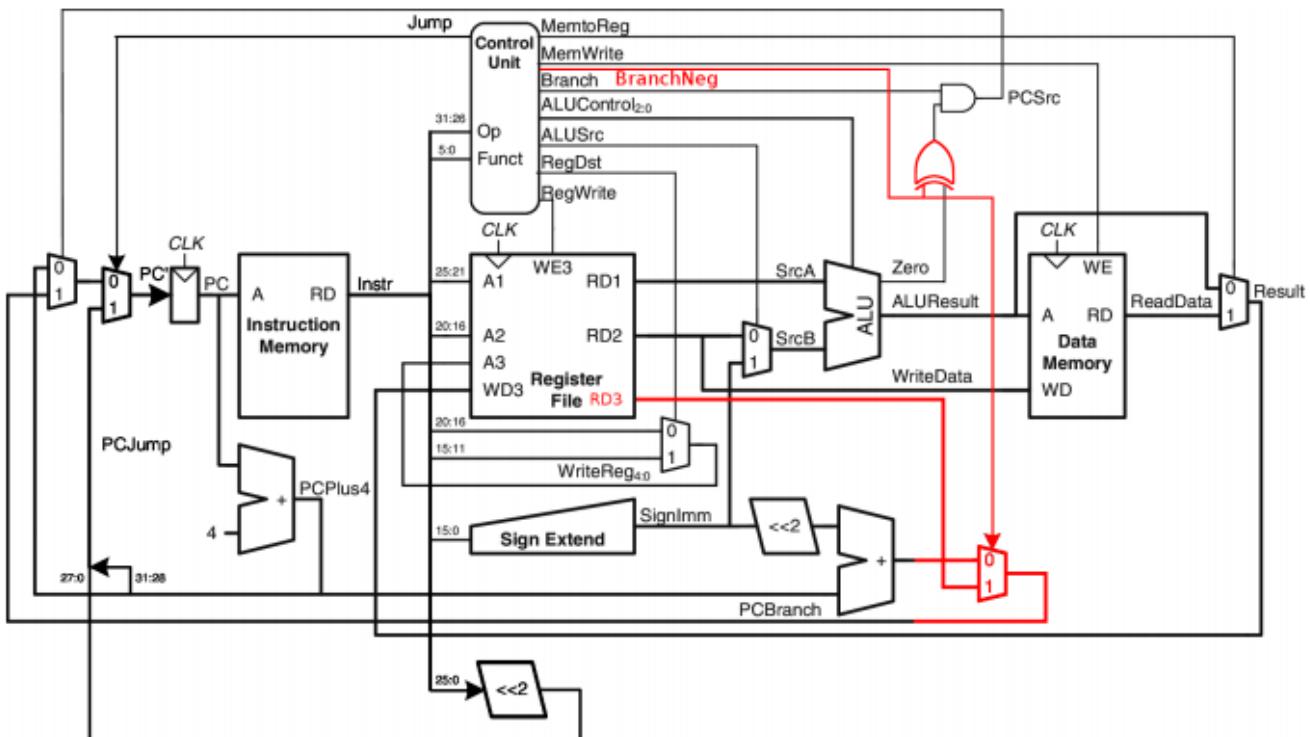


mnemonik	typ	semantyka
min \$Rd,\$Rs,\$Rt	R	Reg[Rd] := (Reg[Rs] < Reg[Rt]) ? Reg[Rs] : Reg[Rt]

	MemToReg	MemWrite	Branch	ALUControl _{2:0}	ALUSrc	RegDst	RegWrite	Jump	Min
min	0/1	0	0	LT	0	1	1	0	1



Zadanie 8 (8). Na poniższym schemacie widnieje jednocyklowa implementacja procesora MIPS. Podaj kodowanie instrukcji, stan sygnałów sterujących i minimalne modyfikacje schematu niezbędne do obsługi dodatkowej instrukcji. Modyfikowanie ALU jest **zabronione!** Do pliku rejestrów można dodać jedno wyjście. Kółeczek na wejściu lub wyjściu bramki oznacza negację sygnału.



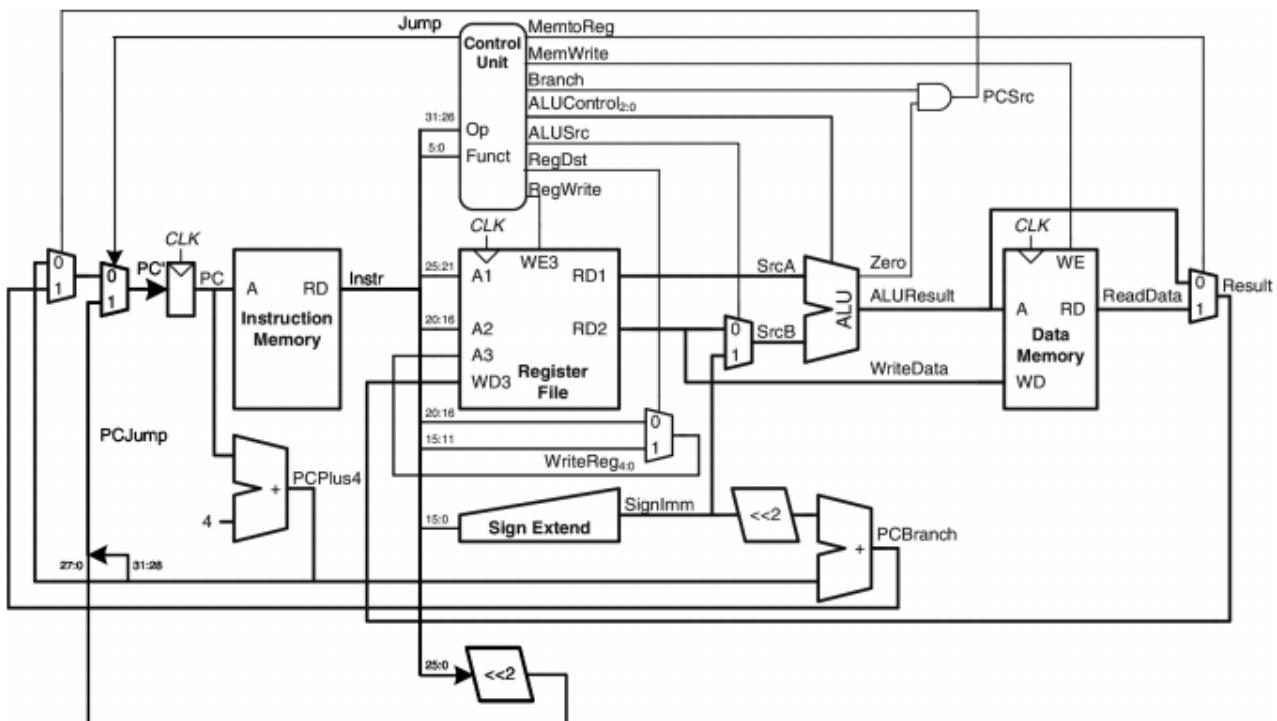
mnemonik	typ	semantyka
brne \$Rs,\$Rt,\$Rd	R	Reg[Rs] != Reg[Rt] => PC := Reg[Rd]

	MemToReg	MemWrite	Branch	ALUControl _{2:0}	ALUSrc	RegDst	RegWrite	Jump	BranchNeg
brne	X	0	1	SUB	0	1	0	0	1

UWAGA: Na schemacie należy jedynie zaznaczyć modyfikowany fragment procesora i przerysować go ze zmianami do kratki! Wszystkie połączenia, z których korzystasz, muszą mieć przypisaną etykietę (np. Result).

Tu należy podać kodowanie instrukcji typu R.

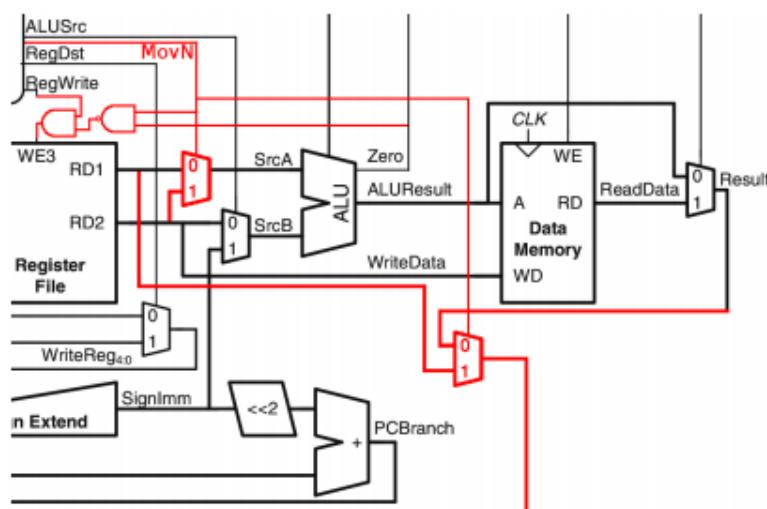
Zadanie 7 (8). Na poniższym schemacie widnieje jednocyklowa implementacja procesora MIPS. Podaj kodowanie instrukcji, stan sygnałów sterujących i minimalne modyfikacje schematu niezbędne do obsługi dodatkowej instrukcji. Modyfikowanie ALU jest **zabronione!** Kóteczko na wejściu lub wyjściu bramki oznacza negację sygnału.



mnemonik	typ	semantyka
movn \$Rd,\$Rs,\$Rt	R	Reg[Rt] ≠ 0 ⇒ Reg[Rd] := Reg[Rs]

	MemToReg	MemWrite	Branch	ALUCtrl _{2:0}	ALUSrc	RegDst	RegWrite	Jump	MovN	
movn	0	0	0	AND	0	1	1	0	1	

UWAGA: Na schemacie należy jedynie zaznaczyć modyfikowany fragment procesora i przrysować go ze zmianami do kratki! Wszystkie połączenia, z których korzystasz, muszą mieć przypisaną etykietę (np. Result).

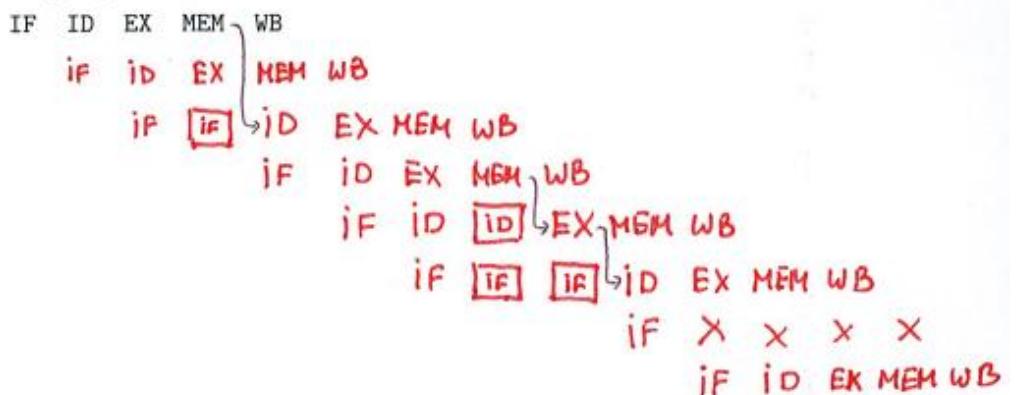


Zadanie 8 (6). Rozważmy potokowy procesor MIPS z implementacją obejścia i obsługą hazardów, ale bez *branch delay slots*. Zakładamy, że skoki są obliczane w fazie ID, a wykonują się w etapie EX. Przyjmujemy strategię przewidywania typu „zawsze nie wykonuj skoku”. Narysuj diagram stanu potoku wykonania poniższego kodu. Wskaż hazardy Read-After-Write i oznacz ścieżki przekazywania danych przy pomocy obejścia. Zakładamy, że skok w linii 3 nie zostaje wykonany, a w linii 6 zostaje.

```

1      lw      $3,0($5)
2      srl    $8,$8,1
3      beq    $2,$3,L1
4      lw      $3,0($6)
5      addi   $3,$3,-16
6 .L1: bne    $2,$3,L2
7      subu   $2,$2,$8
8 .L2: sw      $2,-4($7)

```



Zadanie 8 (10). Rozważmy potokowy procesor MIPS z implementacją obejść i obsługą hazardów, ale bez *branch delay slots*. Zakładamy, że skoki są obliczane w fazie ID, a wykonują się w etapie EX. Przyjmujemy strategię przewidywania typu „zawsze nie wykonuj skoku”. Narysuj diagram stanu potoku wykonania poniższego kodu. Wskaż hazardy *Read-After-Write* i oznacz ścieżki przekazywania danych przy pomocy obejść. Zakładamy, że skok w linii 3 nie zostaje wykonany, a w linii 6 zostaje.

	lw	\$2,8(\$6)	IF	ID	EX	MEM	WB	
	addi	\$6,\$6,32	IF	ID	EX	MEM	WB	
	bne	\$3,\$2,.L1	IF	ID	EX	MEM	WB	
	lw	\$3,0(\$5)	IF	ID	EX	MEM	WB	
	addi	\$3,\$3,-80	IF	ID	EX	MEM	WB	
.L1:	beq	\$3,\$2,.L2	IF	ID	EX	MEM	WB	
	addi	\$7,\$7,-1	IF	ID	EX	MEM	WB	
.L2:	add	\$7,\$2,\$7	IF	ID	EX	MEM	WB	

Zadanie 9 (10). Rozważamy potokowy procesor MIPS z implementacją obejść i obsługi hazardów. Wszystkie skoki obliczane są w fazie ID, a wykonują się w etapie EX. Dla ścieżki, którą wykona się poniższy program, zaznacz w kodzie wszystkie zależności danych i narysuj diagram stanu potoku. Oznacz przekazywanie danych obejściem (strzałką), instrukcje wstrzymane (chmurką) i unieważnione (krzyżykiem).

Zakładamy, że skok w linii 3 nie zostaje wykonany, a w linii 6 za pierwszym razem zostaje, a za drugim nie.

```

j      L2
L1:   lw     $t0, ($a0)
      beq   $t0, $a2, L3
      sw    $t0, 4($a0)
      addi  $a0, $a0, 32
L2:   bne   $a0, $a1, L1
L3:   jr    $ra

```

UWAGA! Opis wykonania jednej instrukcji powinien być umieszczony w jednym wierszu nawet, gdy instrukcja jest wstrzymana.

Instrukcja	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}
j L2		IF	ID	EX	MEM	wB											
lw \$t0, (\$a0)			IF	X	X	X											
bne \$a0, \$a1, L1				IF	ID	EX	MEM	wB									
jr \$ra					IF	X	X	X	X								
lw \$t0, (\$a0)						IF	ID	EX	MEM \searrow	wB							
beq \$t0, \$a2, L3							IF	ID	MEM \nwarrow								
sw \$t0, 4(\$a0)								IF	MEM	wB							
addi \$a0, \$a0, 32									IF	ID	EX \searrow	MEM	wB				
bne \$a0, \$a1, L1										IF	MEM	wB					
jr \$ra											IF	MEM	wB				

Zadanie 8 (10). Rozważamy potokowy procesor MIPS z implementacją obejść i obsługą hazardów. Wszystkie skoki są obliczane w fazie ID, a wykonują się w etapie EX. Dla ścieżki, którą wykona się poniższy program, zaznacz w kodzie wszystkie zależności danych i narysuj diagram stanu potoku. Oznacz przekazywanie danych obejściem (strzałką), instrukcje wstrzymane (chmurką) i unieważnione (krzyżykiem).

Zaczynamy wykonanie programu od linii 4. Należy rozważyć wykonanie instrukcji programu do momentu, w którym licznik rozkazów osiągnął adres etykiety L2. Zakładamy, że skoki w liniach 3 i 7 zostaną wykonane.

```

1: L1: addi $sa0, $a0, 4
2:      lw    $v0, -4($a0)
3:      beq   $v0, $zero, L2
4: addi $sa1, $a1, 4
5:      sw    $v0, -4($a1)
6:      addi $sa2, $a2, -1
7:      bne   $sa2, $zero, L1
8: L2:      jr    $ra

```

UWAGA! Opis wykonania jednej instrukcji powinien być umieszczony w jednym wierszu, nawet gdy instrukcja jest wstrzymana.

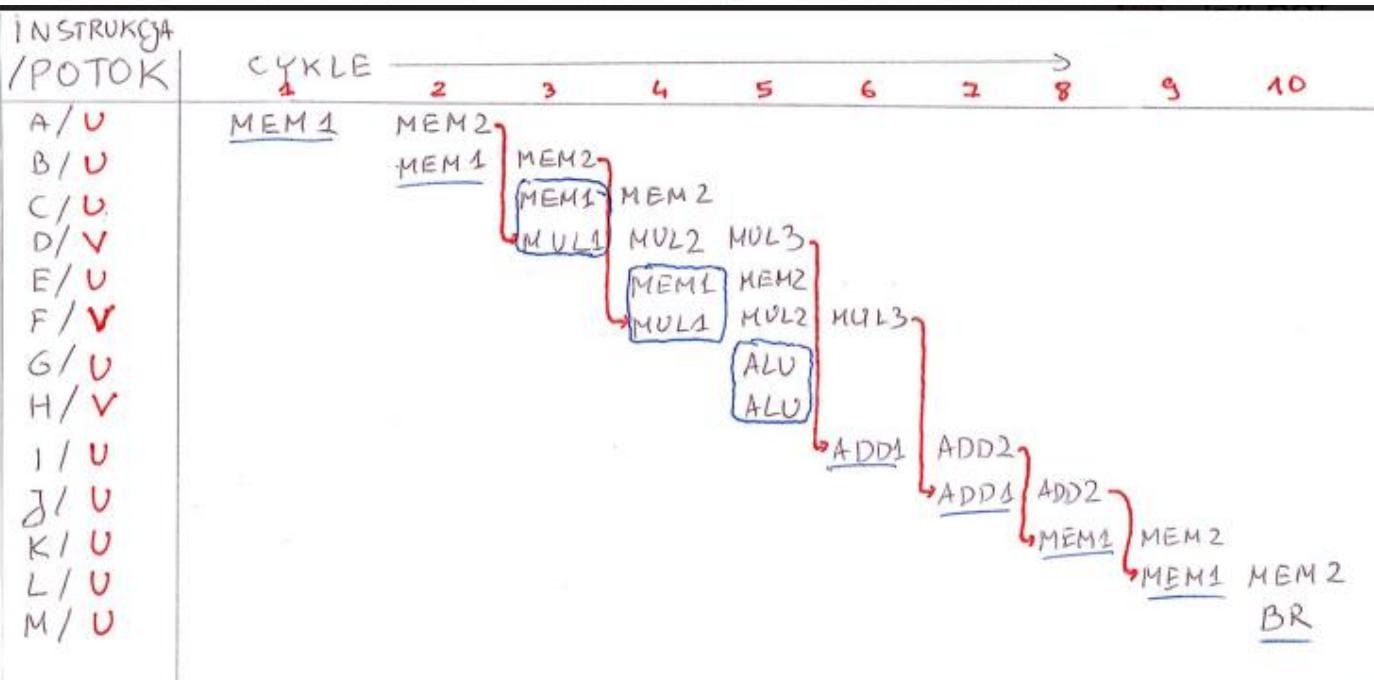
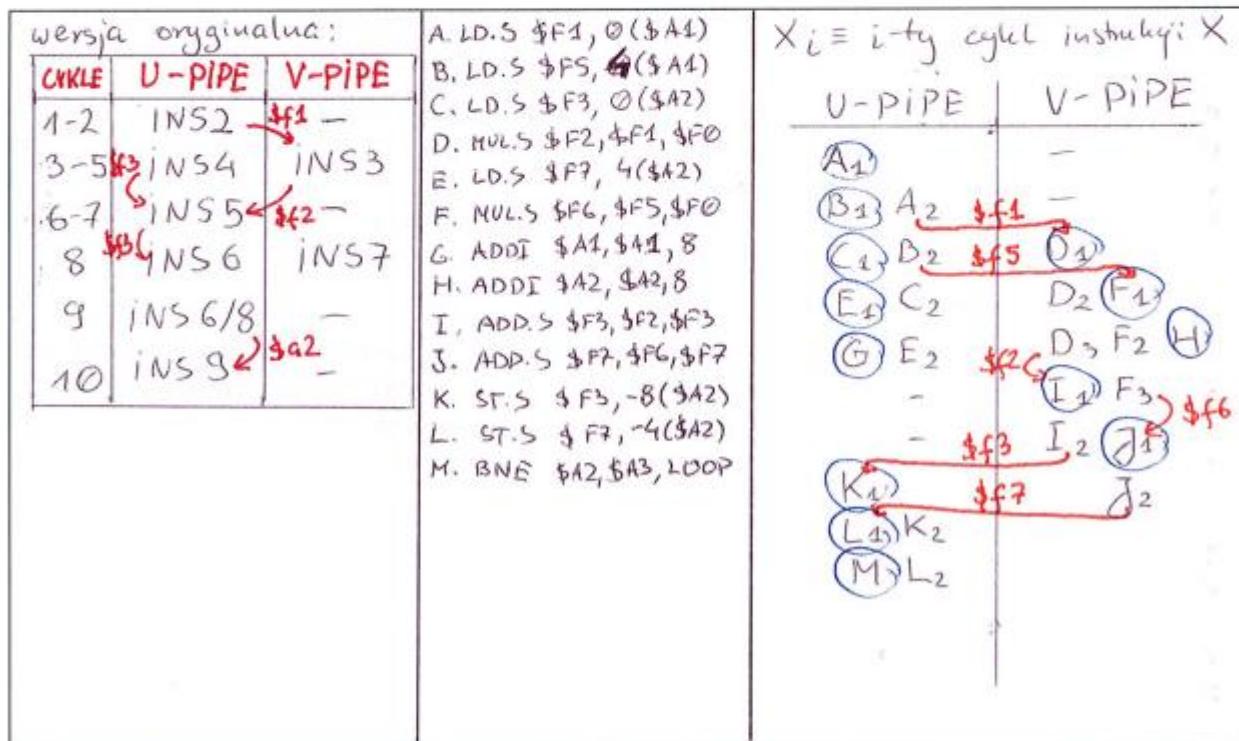
Zadanie 9 (10). Rozważmy superskalarny procesor MIPS z dwoma potokami: U-pipe, który wykonuje wszystkie instrukcje, oraz V-pipe, który wykonuje wyłącznie operacje arytmetyczno-logiczne (w tym na liczbach zmiennopozycyjnych). Czas przetwarzania instrukcji to: ALU - 1, MEM - 2, FP-ADD - 2 i FP-MUL - 3 cykli, a interwał inicjacji wynosi 1. Ile cykli wymaga przetworzenie jednej iteracji poniższej pętli? Rozwiń ją jednokrotnie po czym zoptymalizuj jej ciało, a następnie podaj liczbę cykli wymaganych do jej wykonania.

```

1 loop:
2   ld.s  $f1,0($a1)    # X[i]
3   mul.s $f2,$f1,$f0    # a * X[i]
4   ld.s  $f3,0($a2)    # Y[i]
5   add.s $f3,$f2,$f3    # a * X[i] + Y[i]
6   st.s  $f3,0($a2)    # Y[i] = a * X[i] + Y[i]
7   addi $a1,$a1,8        4
8   addi $a2,$a2,8        4
9   bne   $a2,$a3,loop # $a3 = koniec tablicy Y

```

iteracje	wersja pętli	cykli / element
1x	oryginalna	10
2x	zoptymalizowana	5



Zadanie 16. Mikroarchitektura procesora.

- T Mechanizm przewidywania skoków swym działaniem obejmuje także instrukcje powrotu z procedury.
- V W procesorze potokowym odrębna pamięć podręczna instrukcji i danych całkowicie eliminuje hazard strukturalny w dostępcach do pamięci.
- P Przemianowywanie rejestrów pozwala procesorowi na usuwanie hazardów danych *Read-After-Write*.
- N Procesory *Out-of-Order* pozwalają na zatwierdzania efektów działania instrukcji w innym porządku niż występują one w programie.

Zadanie 12 (5). W jakim celu procesory stosują technikę przewidywania skoków? W jakim etapie przetwarzania instrukcji działa predyktor skoków i jakich informacji potrzebuje?

Przewidywanie skoków stosuje się celom maksymalizacji prędkości przetwarzania instrukcji i minimalizacji opóźnień związanych z przedawnieniem skośników (zarówno warunkowych i bezwarunkowych) w procesorach potokowych i superskalarnych. By procesor przetwarzał wykorzystane następujące instrukcje, tj. pochodzące z faktycznej ścieżki programu, w każdym cyklu fazy IF, CPU musi znać właściwy licznik rozkazów. Niestety prawidłowa wartość PC możliwy poznaje wykorzystując po ukończeniu instrukcji skoku, w związku z tym w połku może znaleźć się instrukcja ze strefy ścieżki w programie. Procesor analizując historię skoków może utrzymywać dwie tablice: PC → Branch Target i PC → Branch Taken? i używać ich w fazie IF do spekulowania, którą ścieżkę pojedzie program. W praktyce obserwuje się, że predyktory skoków podającymi właściwą decyzję w >90% przypadków.

6

Zadanie 12 (6). Podaj główne różnice w przetwarzaniu instrukcji przez procesory potokowe, ze zlecaniem statycznym i dynamicznym. Jakie właściwości biorą pod uwagę kompilatory przeprowadzające szeregowanie instrukcji?

Wymienione procesory dzielą wykonanie instrukcji na etapy. Muszą analizować zależności danych między instrukcjami, aby unikać hazardów danych, kontroli lub sterowania. Różnią się liczbą instrukcji, które mogą zlecić i wykonać w jednym cyklu zegarowym.

Procesor potokowy może zlecić/wykonać maksymalnie jedną instrukcję na cykl. Superskalarny ze zlecaniem statycznym co najwyżej n (liczba równoległych potoków) następnych instrukcji programu. Superskalarny ze zlecaniem dynamicznym, zwany też *Out-of-Order*, co cykl zleca tyle instrukcji ile może wyjąć z kolejki instrukcji i włożyć do poczekalni (ang. *reservation station*). Taki procesor zaczyna wykonywanie instrukcji tak szybko jak zostaną obliczone jej operandy. Nie musi czekać na poprzednie instrukcje, od których nie zależy. W jednym cyklu zegarowym procesor taki może wykonać tyle instrukcji ile ma jednostek wykonawczych (ALU, L/S, BRANCH).

Kompilatory przy szeregowaniu instrukcji biorą pod uwagę zależności danych i kontroli, interwał inicjacji (co ile cykli można rozpoczęć wykonanie takiego samego typu instrukcji – dla dzielenia np. 20), opóźnienie przetwarzania (po ilu cyklach dostaniemy wynik operacji – dla mnożenia np. 3, optymistycznie dostęp do pamięci np. 4), zajętość jednostek wykonawczych danego typu w danym cyklu zegarowym.

Zadanie 14 (4). Mikroarchitektura procesora.

- N Hazard kontroli występuje, gdy przetwarzanie instrukcji wymaga użycia tych samych zasobów sprzętowych.
- T Mechanizm przewidywania skoków obsługuje zarówno skoki warunkowe jak i bezwarunkowe.
- T Przemianowywanie rejestrów pozwala procesorowi na usuwanie zależności danych typu *Write-After-Read*.
- N Procesor *Out-of-Order* może zatwierdzać efekty działania instrukcji w porządku innym niż wynikający z kolejności rozkazów w programie.