

Architektura počítačů a operačních systémů

Studijní materiál ke státnicím (I4)

Jakub Klesa

Pokryté předměty: NSWI120 (Principy počítačů), NSWI170 (Počítačové systémy)

Obsah

Úvod	2
1 Architektura počítače, reprezentace čísel, dat a programů	3
1.1 Základní architektura počítače	3
1.2 Paměť, adresa a adresový prostor	3
1.3 Reprezentace celých čísel	4
1.4 Reprezentace reálných čísel (IEEE 754)	4
1.5 Základní aritmetické a logické operace	4
1.6 Ukládání složených typů a zarovnání	5
1.7 Řešený příklad: reprezentace dat v binárním souboru	5
1.8 Dynamická správa paměti na haldě	7
2 Instrukční sada a vazba na vyšší programovací jazyky	8
2.1 Instrukce a instrukční sada	8
2.2 Třídy instrukcí a registry	9
2.3 Mapování konstrukcí vyššího jazyka na instrukce	10
2.4 RISC vs. CISC	12
2.5 Řešené příklady	12
3 Podpora běhu operačního systému	13
3.1 Role operačního systému	13
3.2 Privilegovaný a neprivilegovaný režim procesoru	13
3.3 Přejít mezi režimy a systémové volání	14
3.4 Jádro operačního systému	14
3.5 Řešený příklad: které operace vyžadují jádrový režim	15
4 Rozhraní periférií a jejich obsluha	16
4.1 Zařízení, řadič a ovladač	16
4.2 Jak procesor komunikuje se zařízením	16
4.3 Programem řízená obsluha (PIO) s aktivním čekáním	17
4.4 Přerušování	17
4.5 Reakce na žádost o přerušování (hardware + software)	18
4.6 Řešený příklad: ovladač pro řadič disku	19
5 Abstrakce OS: procesy, vlákna, plánování a soubory	21
5.1 Proces a vlákno	22
5.2 Kontext a přepínání kontextu	22
5.3 Multitasking: kooperativní vs. preemptivní	22
5.4 Plánování (scheduling)	23

5.5	Stavy plánovací jednotky	24
5.6	Adresový prostor procesu (kde leží zásobníky vláken)	24
5.7	Soubory: sdílení úložného prostoru	25
5.8	Řešený příklad: plánování Round Robin	26
6	Virtuální paměť a překlad adres	27
6.1	Virtuální vs. fyzická adresa	27
6.2	Stránkování: stránky, rámce, stránkovací tabulka	27
6.3	Překlad adresy nad jednoúrovňovou tabulkou	28
6.4	TLB: urychlení překladu	29
6.5	Ochrana paměti	30
7	Paralelismus a synchronizace	32
7.1	Paralelismus a souběžnost	32
7.2	Časově závislé chyby (race conditions)	32
7.3	Kritická sekce a vzájemné vyloučení	32
7.4	Synchronizační primitiva	33
7.5	Řešené příklady	33
	Shrnutí	36
	Zdroje	39

Úvod

Tento materiál pokrývá okruh **I4 – Architektura počítačů a operačních systémů** (předměty NSWI120 *Principy počítačů* a NSWI170 *Počítačové systémy*). Výklad stojí na oficiálních přednáškových materiálech obou kurzů z MFF UK: na slidech NSWI170 (J. Yaghob, M. Kruliš), které pokrývají hlavně část o operačních systémech (režimy procesoru, jádro, periférie, procesy a plánování, virtuální paměť, synchronizace), a na slidech NSWI120 (P. Ježek, slidová sada L. Buleje) pro nízkoúrovňovou reprezentaci dat a instrukční sadu. Tam, kde slidy jen heslují (dvojkový doplněk, IEEE 754, mapování konstrukcí vyššího jazyka na instrukce), doplňujeme výklad volně dostupnou učebnicí *Dive Into Systems* a ověřujeme jej standardní referencí. Terminologii i hloubku držíme na úrovni bakalářských kurzů.

Role důkazů. Požadavky okruhu *nejmenují žádné věty ani důkazy*. Jde o definice, popis mechanismů a jejich aplikaci na konkrétní zadání: přeložit virtuální adresu stránkovací tabulkou, dekodovat hexdump podle popsaného formátu, naprogramovat obsluhu zařízení nebo synchronizaci přes semaforey, najít v kódu časově závislou chybu. Místo důkazu se nanejvýš *zdůvodňuje* (proč v daném kódu nastane race condition, proč nemůže nastat deadlock).

Do výkladu je zařazeno všech deset reálných otázek SZZ k okruhu z let 2022–2026 jako řešené příklady (tři z nich, z léta 2023, formálně patřily jiné specializaci); jsou i v samostatném dokumentu *Příklady otázek* (tam i s tabulkami oříznutými z PDF). U úloh, které v originálním PDF nemají náčrt řešení, jsme napsané řešení nezávisle ověřili simulací v Pythonu. Použité zdroje jsou uvedeny na konci.

Značení. Bit je nejmenší jednotka, osm bitů tvoří *bajt* (byte), který je zároveň jednotkou adresace. Číselné soustavy odlišujeme prefixem: bez prefixu desítkově, 0x šestnáctkově, 0b dvojkově (např. $0x2C = 44 = 0b101100$). Šířku slova procesoru značíme N bitů (typicky 32 nebo 64). *Virtuální adresu* píšeme VA, *fyzickou* FA; u stránkování značíme číslo stránky VPN (*virtual page number*), číslo rámce PFN (*page frame number*) a posunutí v rámci stránky *offset*. Pokud neuvedeme jinak, předpokládáme *little-endian* uložení vícebajtových čísel (jako x86). Kód sázíme strojopisem ve výpisech (*listings*); komentáře v kódu jsou kvůli sazbě anglicky, prozaický výklad kolem je česky.

1 Architektura počítače, reprezentace čísel, dat a programů

Požadavky chtějí umět popsat základní stavbu počítače a hlavně *reprezentaci a přístup k datům v paměti*: jak vypadá adresa a adresový prostor, jak jsou v paměti uloženy jednoduché i složené datové typy a jak nad nimi pracují základní aritmetické a logické operace. Tato sekce zavádí pojmy, na kterých staví celý zbytek okruhu (adresy a offsety u stránkování, hexdumpy u obsluhy zařízení, zarovnání u správce paměti).

1.1 Základní architektura počítače

Definice – Von Neumannova vs. harvardská architektura

Von Neumannova architektura ukládá program i data do *téže* paměti, přístupné po jedné sběrnici. *Harvardská architektura* má oddělené paměti (a sběrnice) pro instrukce a pro data. Běžné PC jsou von Neumannova; harvardská se používá v mikrokontrolérech a uvnitř procesoru na úrovni cache (oddělené L1 pro instrukce a data).

Procesor (CPU) je stroj, který opakovaně vykonává jednoduché *instrukce*. Adresu právě prováděné instrukce drží registr *IP* (instruction pointer, též PC, program counter).

Definice – Instrukční cyklus

Vykonání jedné instrukce probíhá v krocích: *načti* instrukci z adresy v IP (fetch), *dekóduj* ji (decode), *načti operandy*, *proved'* operaci (execute), *ulož* výsledek a *posuň* IP na následující instrukci. U některých instrukcí se některé kroky vynechají.

 **Intuice.** Celý výpočet je posloupnost těchto cyklů. Procesor sám o sobě „nerozumí“ cyklům ani funkcím vyššího jazyka – ty se na instrukce teprve překládají (sekce 2). Význam má jen aktuální obsah registrů a paměti a adresa v IP.

1.2 Paměť, adresa a adresový prostor

Definice – Paměť, bajt, adresa, adresový prostor

Paměť je tvořena *buňkami* (bity) sdruženými do *slov* pevné délky. Dnes je jednotkou adresace *bajt* (byte, 8 bitů): každý bajt má vlastní *adresu*, což je N -bitové číslo (typicky $N = 32$ nebo 64). Množina všech adres $[0, 2^N)$ je *adresový prostor*; lze v něm adresovat 2^N bajtů. Hodnota uložená v ukazateli (pointeru) je právě taková adresa.

Jednoduchá data (celé číslo, znak, ukazatel) zabírají souvislou skupinu bajtů začínající na nějaké adrese. U vícebajtových hodnot je ale nutné dohodnout *pořadí bajtů* v paměti.

Definice – Endianita (pořadí bajtů)

Little-endian: na nejnižší adresu se uloží nejméně významný bajt (LSB), na nejvyšší nejvýznamnější (MSB). *Big-endian*: opačně, na nejnižší adresu nejvýznamnější bajt. Intel x86 je little-endian; síťové protokoly TCP/IP používají big-endian (proto „network byte order“).

	A	A+1	A+2	A+3	
big-endian:	0A	0B	0C	0D	MSB na nejnižší adrese
little-endian:	0D	0C	0B	0A	LSB na nejnižší adrese

Uložení 32-bitové hodnoty $0x0A0B0C0D$ od adresy A . Bajty hodnoty jsou (od nejvýznamnějšího) $0A\ 0B\ 0C\ 0D$; liší se jen jejich pořadí v paměti. Čtení čísla zpět musí použít stejnou konvenci, jakou bylo zapsáno.

1.3 Reprezentace celých čísel

Definice – Bez znaménka a dvojkový doplněk

N -bitové číslo *bez znaménka* (unsigned) je prostý dvojkový zápis; rozsah je $[0, 2^N - 1]$. Číslo se *znaménkem* se ukládá ve *dvojkovém doplňku*: nejvyšší bit má váhu -2^{N-1} , ostatní kladné váhy. Rozsah je $[-2^{N-1}, 2^{N-1} - 1]$ (asymetrický). Opačnou hodnotu získáme bitovou negací a přičtením jedničky: $-x = \sim x + 1$.

 **Intuice.** Dvojkový doplněk má jen *jednu* nulu (na rozdíl od reprezentace „znaménko + velikost“) a hlavně se sčítá a odčítá *stejným* obvodem jako čísla bez znaménka – bity výsledku jsou identické, liší se jen jejich interpretace. Proto má procesor jednu sčítačku pro obě varianty. Např. v 8 bitech je $0xFF = 255$ bez znaménka, ale -1 se znaménkem; $0x80 = 128$ resp. -128 .

1.4 Reprezentace reálných čísel (IEEE 754)

Reálná čísla se ukládají v *pohyblivé řádové čárce* (floating point) podle normy IEEE 754. Jednoduchá přesnost (**float**, 32 bitů) má 1 bit znaménka, 8 bitů exponentu a 23 bitů mantisy (zlomkové části).

31	30 ... 23	22 ... 0
s	exponent e (8 b)	mantisa f (23 b)

Formát IEEE 754 single (32 b). Hodnota normalizovaného čísla je $(-1)^s \cdot (1.f)_2 \cdot 2^{e-127}$, kde exponent je uložen s posunem (bias) 127 a před mantisou se předpokládá implicitní jednička. Např. $1.0 = (-1)^0 \cdot 1.0 \cdot 2^{127-127}$, tj. $s=0, e=127=0x7F, f=0$, dohromady $0x3F800000$.

 **Poznámka.** Krajní hodnoty exponentu jsou vyhrazené: samé nuly kódují nulu a *denormalizovaná* čísla (velmi malá), samé jedničky kódují $\pm\infty$ (mantisa 0) a *NaN* (mantisa $\neq 0$). Dvojitá přesnost (**double**, 64 b) má stejnou strukturu s 11 bity exponentu a 52 bity mantisy.

1.5 Základní aritmetické a logické operace

Procesor poskytuje *aritmetické* instrukce (+, −, ×, celočíselné / a zbytek %) a *bitové/logické* instrukce: bitové a (&), *nebo* (|), *xor* (^), *negace* (−) a *posuny*.

Definice – Posuny a přetečení

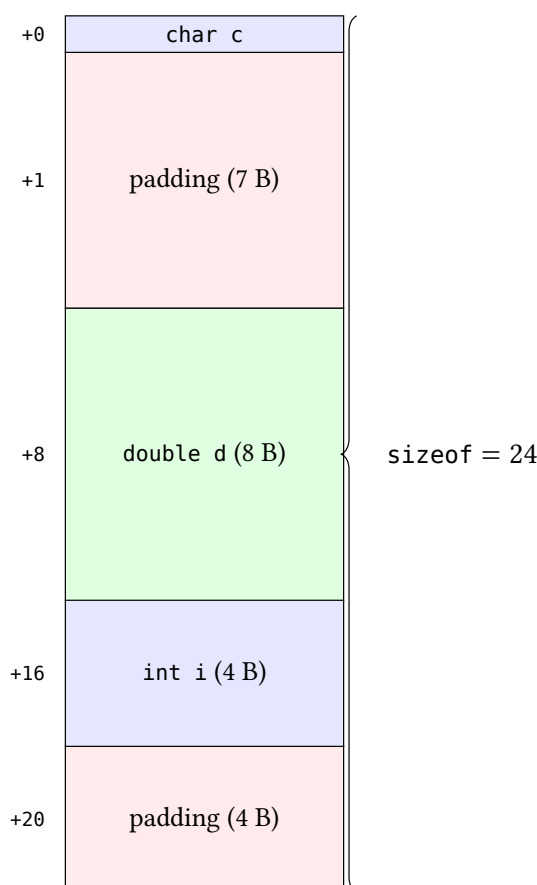
Logický posun (<<, >> na unsigned) doplňuje nuly; *aritmetický posun vpravo* (na signed) doplňuje znaménkový bit, takže zachová znaménko. Posun vlevo o k je násobení 2^k , posun vpravo dělení 2^k (dolů). *Přetečení* (overflow): u čísel bez znaménka výsledek „přeteče“ modulo 2^N ; u znaménkových je přetečení, když pravý matematický výsledek leží mimo rozsah dvojkového doplňku.

1.6 Ukládání složených typů a zarovnání

Složené typy (struktury, pole) jsou poskládané z jednodušších, ale ne libovolně těsně.

Definice – Zarovnání a výplň (padding)

Zarovnání (alignment) datového typu velikosti s znamená, že jeho adresa musí být dělitelná s (procesory to pro efektivní přístup vyžadují, příp. silně preferují). Struktura se zarovnává podle svého *největšího* členu. Aby každý člen padl na svou zarovnanou adresu, vkládá překladač mezi členy nevyužité bajty – *vnitřní výplň* (inner padding) – a na konec struktury *vnější výplň* (outer padding) tak, aby i v poli za sebou jdoucí prvky zůstaly zarovnané, tj. velikost struktury je násobkem jejího zarovnání.



Rozložení struct { char c; double d; int i; } v paměti (64-bit, zarovnání struktury = 8 podle double). c je na offsetu 0, ale d musí začínat na násobku 8, proto je za c vnitřní výplň 7 B. Za i (offsety 16–19) je vnější výplň 4 B, aby celková velikost 24 byla násobkem 8. Holá data zabírají $1+8+4 = 13$ B, struktura ale 24 B.

1.7 Řešený příklad: reprezentace dat v binárním souboru

Příklad – SZZ podzim 2025 č. 2: Reprezentace dat v binárním souboru

(Reálná otázka SZZ: podzim 2025, č. 2 – Ukládání dat v binárním souboru (hexdump, reprezentace dat))

Zadání.

Soubor ukládá uzly se stejnou sadou atributů. Hlavička na začátku obsahuje 2 bajty s počtem atributů, pak po jednom bajtu typ každého atributu (enum AttrType { TUInt16=0x48, TLink=0x4C, TString=0x53, TUInt16=0x68 }). Vícebajtová čísla jsou **big-endian**. Hned za hlavičkou je kořenový uzel: 2 bajty délky dat, pak hodnoty atributů. TUInt16/TUInt16 jsou 2 bajty (bez/se znaménkem, dvojkový doplněk), TString je 2 bajty délky + ASCII znaky, TLink je 8 bajtů (pozice jiného uzlu). Třída BinaryFile nabízí

`void readBytes(long filepos, byte[] buf, int len)`. Úkolem je napsat třídu Reader (konstruktor načte hlavičku; attributes, getType, getRoot, readNode) a třídu Node (data jako pole bajtů + struktura pro lokalizaci atributu v konstantním čase; metody getUInt16, getSInt16).

Řešení.

(1) **Dekódování ukázkového hexdumpu.** Nejdřív si rozmysleme, jak se daný výpis čte – to je jádro úlohy. Bajty jsou (offsety vlevo):

```
0000: 00 06 48 68 53 53 4C 4C  <- hlavička: 0x0006 = 6 atributu, pak typy
0008: 00 22 03 E8 FC 18 ...  <- kořenový uzel: 0x0022 = 34 B dat
```

Typy atributů jsou 0x48, 0x68, 0x53, 0x53, 0x4C, 0x4C, tj. TUInt16, TSInt16, TString, TString, TLink, TLink. Kořenový uzel začíná na offsetu 8 (za 2 B počtu a 6 B typů); jeho data (34 B) se čtou podle typů:

1. TUInt16: 03 E8 big-endian = $0x03E8 = 1000$.
2. TSInt16: FC 18 = $0xFC18$; nejvyšší bit 1, tedy záporné: $0xFC18 - 2^{16} = -1000$.
3. TString: délka 00 05 = 5, pak 48 65 6C 6C 6F = "Hello".
4. TString: délka 00 05 = 5, pak 57 6F 72 6C 64 = "World".
5. TLink: 8 bajtů 00 00 00 00 00 00 00 2C = $0x2C$.
6. TLink: 8 bajtů 00 00 00 00 00 00 12 34 = $0x1234$.

Spotřebovaná délka je $2 + 2 + (2+5) + (2+5) + 8 + 8 = 34$ B, což přesně sedí s deklarovanou délkou uzlu.

(2) **Třída Reader.** Konstruktor přečte počet atributů a jejich typy; getRoot vrací offset hned za hlavičkou. readNode přečte 2 bajty délky a pak len bajtů dat. Pomocné big-endian čtení 16-bit čísla:

```
struct Reader {
    BinaryFile& f; int nattr; std::vector<unsigned char> types;
    Reader(BinaryFile& bf) : f(bf) {
        unsigned char h[2]; f.readBytes(0, h, 2);
        nattr = (h[0] << 8) | h[1]; // big-endian
        types.resize(nattr); f.readBytes(2, types.data(), nattr);
    }
    int attributes() { return nattr; }
    int getType(int i) { return types[i]; }
    long getRoot() { return 2 + nattr; } // hned za hlavičkou
    Node readNode(long filepos) {
        unsigned char len2[2]; f.readBytes(filepos, len2, 2);
        int len = (len2[0] << 8) | len2[1];
        std::vector<unsigned char> data(len);
        f.readBytes(filepos + 2, data.data(), len);
        return Node(types, std::move(data));
    }
};
```

(3) **Třída Node.** Data drží jako pole bajtů; aby šel i -tý atribut najít v $O(1)$, předpočítá v konstruktoru pole offsetů každého atributu uvnitř dat (řetězce mají proměnnou délku, proto se offsety musí jednou projít). Převod na číslo dělají get metody pomocí big-endian skládání:

```
struct Node {
    std::vector<unsigned char> data; std::vector<int> off; // off[i] = pozice atributu i
    Node(const std::vector<unsigned char>& types, std::vector<unsigned char> d)
        : data(std::move(d)) {
        int p = 0;
        for (unsigned char t : types) {
            off.push_back(p);
            if (t == 0x48 || t == 0x68) p += 2; // TUInt16 / TSInt16
            else if (t == 0x4C) p += 8; // TLink
            else if (t == 0x53) p += 2 + ((data[p]<<8)|data[p+1]); // TString
        }
    }
};
```

```
int getUInt16(int i) { int p=off[i]; return (data[p]<<8) | data[p+1]; }
int getSInt16(int i) { int v=getUInt16(i); return v>=0x8000 ? v-0x10000 : v; }
};
```

Pozn.: V PDF tato otázka náčrt řešení nemá; dekodování i kód jsme ověřili skriptem `verify_szz.py` (hodnoty {1000, -1000, "Hello", "World", 0x2C, 0x1234}).

1.8 Dynamická správa paměti na haldě

Lokální proměnné a argumenty žijí na *zásobníku* (alokace posunem ukazatele, sekce 2), globální mají pevné místo. Proměnné, jejichž životnost neurčuje rozsah bloku, se alokují dynamicky na *haldě* (heap): běhové prostředí (resp. OS) spravuje velký blok paměti a přiděluje z něj kusy na žádost.

Intuice. Dynamická alokace není v doslovných požadavcích okruhu, ale objevuje se v SZZ otázce léto 2025 a přesně procvičuje reprezentaci dat v paměti, zarovnání a práci s adresami a offsety – proto ji řešíme jako příklad. Správce vede *evidenci volných bloků* (typicky vázaný seznam) a na žádost vybere vhodný blok strategií *first-fit* (první dost velký od začátku), případně *best/next/worst-fit*. Volné místo se časem tříští (*fragmentace*).

Příklad — SZZ léto 2025 č. 2: Jednoduchý správce paměti

(Reálná otázka SZZ: léto 2025, č. 2 – Jednoduchý správce paměti (heap alokátor, 32-bit))

Zadání.

V souvislém poli bajtů (little-endian) implementujeme heap. Každý *podblok* je zarovnaný na 2 bajty a má 16-bitovou hlavičku *Size* na offsetu +0 = velikost podbloku *včetně hlavičky*. Protože velikost je sudá, je její bit 0 vždy 0 a používá se jako příznak: 0 = volný, 1 = obsazený. Volný podblok má navíc na +2 položku *Next* (16-bit offset dalšího volného podbloku, -1 = 0xFFFF u posledního), zbytek je vynulovaný; volné podbloky tak tvoří jednosměrný seznam. Obsazený podblok má na +2 rovnou *payload*. Pamatujeme si offset prvního volného podbloku; na počátku je heap jeden volný podblok velikosti celého heapu. Alokuje se strategií *first-fit*. Dány metody *SetUshort/GetUshort* (čtení/zápis 16-bit hodnoty little-endian). Úkoly: (1) konstruktor, (2) *FindFirstFree*, (3) *Mark*, (4) *hexdump* 22 B heapu po `A=Alloc(2); B=Alloc(4); C=Alloc(2); Free(B)`.

Řešení.

(1)–(3) Metody. Potřebná velikost podbloku je `payload + 2` (hlavička), zaokrouhleno na sudé číslo. Konstruktor založí jediný volný podblok přes celý heap (a podle zadání vynuluje jeho zbytek, paměť nemusí přijít vynulovaná); *FindFirstFree* projde seznam volných a vrátí první dost velký; *Mark* jen přepíše bit 0 položky *Size*.

```
public Heap(byte[] availableMemory) {
    _heap = availableMemory;
    _firstFreeOffset = 0;
    SetUshort(0, (ushort)_heap.Length); // Size = cely heap, bit0=0 (volny)
    SetUshort(2, 0xFFFF);                // Next = -1 (posledni volny)
    for (int i = 4; i < _heap.Length; i++) // zbytek volneho podbloku nulovy
        _heap[i] = 0;
}

private ushort FindFirstFree(ushort payloadSize) {
    ushort need = (ushort)(payloadSize + 2);
    if ((need & 1) != 0) need++;           // zarovnani na 2
    ushort o = _firstFreeOffset;
    while (o != 0xFFFF) {
        ushort size = (ushort)(GetUshort(o) & 0xFFFE); // bit0 je priznak
        if (size >= need) return o;
        o = GetUshort((ushort)(o + 2));           // Next
    }
    return 0xFFFF;                             // nenalezeno
}
```

```

}
private void Mark(ushort offset, bool isFree) {
    ushort size = GetUshort(offset);
    SetUshort(offset, isFree ? (ushort)(size & 0xFFFE)
                             : (ushort)(size | 1));
}

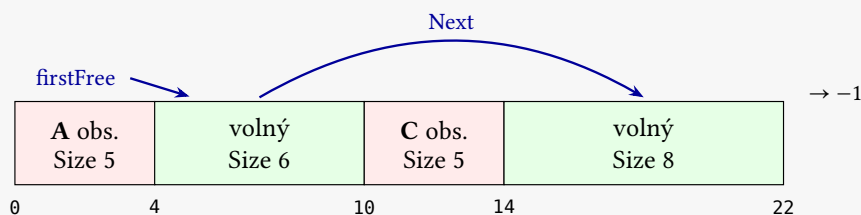
```

(4) **Stav heapu krok po kroku** (heap má 22 B, na počátku samé nuly, jeden volný podblok velikosti 22 s `Next = -1`):

1. `Alloc(2)` → potřeba $2+2 = 4$. First-fit vezme volný blok @0 ($22 \geq 4$), rozdělí: **A** obsazený @0 velikosti 4, zbytek volný @4 velikosti 18. Payload A na @2.
2. `Alloc(4)` → potřeba $4+2 = 6$. Volný @4 ($18 \geq 6$): **B** obsazený @4 velikosti 6, zbytek volný @10 velikosti 12. Payload B na @6.
3. `Alloc(2)` → potřeba 4. Volný @10 ($12 \geq 4$): **C** obsazený @10 velikosti 4, zbytek volný @14 velikosti 8. Payload C na @12.
4. `Free(B)`: payload B byl na @6, podblok tedy @4; označí se volným. Do seznamu volných se vloží podle offsetu: první volný je nyní @4 (`Next` → 14), pak @14 (`Next` = -1).

Výsledné podbloky: A obsazený @0 (Size 5), volný @4 (Size 6, `Next` 14), C obsazený @10 (Size 5), volný @14 (Size 8, `Next` -1). Hexdump (16-bit hodnoty little-endian; Size obsazeného je velikost | 1):

offset	+0	+1	+2	+3	+4	+5	+6	+7
0x0000	05	00	00	00	06	00	0E	00
0x0008	00	00	05	00	00	00	08	00
0x0010	FF	FF	00	00	00	00	--	--



Stav heapu po `Alloc(2)/Alloc(4)/Alloc(2)/Free(B)`. Obsazené podbloky A, C (červeně) zůstaly, uvolněný B se stal volným podblokem @4. Volné podbloky tvoří seznam `firstFree` → @4 → @14 → -1. U obsazených bloků je v poli `Size` nastaven příznakový bit 0, proto se reálná velikost 4 uloží jako `Size 5 = 4 | 1`. Hexdump výše ověřen simulací (`verify_szz.py`).

2 Instrukční sada a vazba na vyšší programovací jazyky

Procesor sám neumí cykly ani funkce – vykonává jen jednoduché *instrukce*. Tato sekce ukazuje, co je instrukční sada, jaké třídy instrukcí existují, a hlavně *jak se běžné konstrukce vyššího jazyka* (přiřazení, podmínka, cyklus, volání funkce) překládají na instrukce a jak naopak ze sekvence instrukcí poznat původní konstrukci. To je jádro doslovných požadavků k této části okruhu. Assembler píšeme symbolicky ve stylu MIPS (jak ho používají slidy NSWI120/NSWI170); jde o čitelný zápis instrukcí, ne o přesnou syntaxi konkrétního procesoru.

2.1 Instrukce a instrukční sada

Definice – Instrukční sada (ISA)

Instrukce je jednoduchý příkaz pro procesor (jedna aritmetická či logická operace, přesun dat, skok). *Instrukční sada* (instruction set) je seznam všech instrukcí, kterým daný procesor rozumí. *Architektura instrukční sady* (ISA, *instruction set architecture*) je její abstraktní specifikace – *kontrakt* mezi autorem programu (resp. překladačem) a návrhářem hardwaru: říká, jaké instrukce existují,

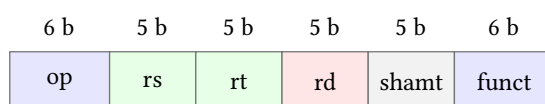
jaké mají operandy, jaké jsou registry a jak se kódují. Tentýž kontrakt může splňovat víc různých implementací hardwaru.

💡 Intuice. Instrukční sada je „slovník“ jazyka procesoru, jednotlivé instrukce jsou „slova“. Vyšší programovací jazyk (C, C++) je proti tomu výrazně bohatší; mezi nimi je *sémantická mezera* (semantic gap), kterou překlene *překladač*: rozloží složité konstrukce na posloupnost těchto jednoduchých slov. Formálně stačí překvapivě málo instrukcí na vykonání libovolného výpočtu; o zbytku rozhodují praktické ohledy (jednoduchost hardwaru, rychlost, srozumitelnost).

Operandy instrukce jsou většinou *registr* (rychlé paměťové místo přímo v procesoru) nebo *přímý operand* (immediate, konstanta zakódovaná přímo v instrukci); přístup do hlavní paměti dělají jen vyhrazené instrukce (load/store). Počet operandů je dán ISA – typicky 0 až 3, např. aritmetika MIPS má vždy 3 (dva zdroje + cíl).

📖 Definice – Strojový kód vs. assembler; délka instrukce

Procesor čte instrukce jako *strojový kód*: posloupnost bitů, kde jedna část kóduje operaci (*opcode*) a další části operandy (čísla registrů, konstantu). Pro člověka se totéž zapisuje *assemblerem* – symbolickými jmény (add, lw, jména registrů, návěští). Instrukce mohou mít *pevnou délku* (např. MIPS: každá instrukce 4 bajty – jednoduché dekodování) nebo *proměnnou délku* (např. x86: 1 až 15 bajtů – kompaktnější kód, složitější dekodování).



add \$t0,\$s1,\$s2 → 0 | 17 | 18 | 8 | 0 | 32

Pevný 32-bitový formát registrové (r-type) instrukce MIPS: opcode, dva zdrojové registry rs, rt, cílový rd, velikost posunu shamt a varianta operace funct. Assembler add \$t0,\$s1,\$s2 se přeloží na čísla 0 | 17 | 18 | 8 | 0 | 32 (registry \$s1, \$s2 mají čísla 17, 18, \$t0 číslo 8). Vše se vejde do 32 bitů, proto má každá instrukce stejnou délku.

2.2 Třídy instrukcí a registry

📖 Definice – Hlavní třídy instrukcí

Přesuny dat: load (paměť → registr, lw), store (registr → paměť, sw), move (registr → registr). Aritmetické operace pracují (u load-store architektur jako MIPS) jen s registry, proto se data nejdříve načtou a výsledek uloží zpět. *Aritmetické a logické*: +, −, ×, /, % a bitové &, |, ^, ~, posuny <<, >>. *Skoky* (branch/jump): *nepodmíněné* (j) a *podmíněné* (beq, bne – skok podle výsledku testu); podle cíle *přímé* (adresa v instrukci), *nepřímé* (adresa v registru) a *relativní* (posun vůči současné adrese). *Volání a návrat*: call (jal) a ret (jr) pro funkce.

💡 Poznámka. ISA bez příznaků (jako MIPS) řeší porovnání instrukcí typu slt (set-on-less-than): zapíše do registru 0/1 následovaným beq/bne. ISA s příznaky (x86) má univerzální cmp, které nastaví *příznakové bity*, a podmíněné skoky (jl, jz, ...) se rozhodují podle nich. Obě cesty realizují totéž rozhodnutí.

📖 Definice – Registry a příznaky

Registr je paměťové místo přímo v procesoru (přístup mnohem rychlejší než do paměti, ale je jich málo – jednotky až desítky). Bývají *obecné* (data, adresy), speciální jako IP/PC (instruction pointer /

program counter – adresa právě prováděné instrukce) a na některých ISA *příznakový registr* (flags). Obvyklé příznaky: Z (zero – výsledek byl 0), S (sign – výsledek záporný), C (carry – přenos/výpůjčka u operací bez znaménka), O (overflow – přetečení u operací se znaménkem).

Protože proměnných je v programu mnohem víc než registrů, drží překladač v registrech nejčastěji používané hodnoty a ostatní podle potřeby přesouvá do paměti a zpět (*register spilling*).

2.3 Mapování konstrukcí vyššího jazyka na instrukce

Toto je jádro sekce. Postupně projdeme čtyři konstrukce z požadavků a ukážeme *oba směry*: jazyk → instrukce i zpět.

Přiřazení. Výraz na pravé straně se vyhodnotí do registru a výsledek se zapíše do cílové proměnné. Konstanta jde přímo jako immediate, posun je jediná instrukce:

```
b = 4;           // b is a register/variable -> use a constant
li    r1, 4      // r1 = 4 (load immediate)
store r1, [b]    // write r1 into b

c = a << 2;      // shift = single instruction, no real multiply
load  r1, [a]    // r1 = a
shl   r1, r1, 2   // r1 = a << 2 (i.e. a * 4)
store r1, [c]    // write r1 into c
```

Vyhodnocení složitějšího výrazu rozdělí překladač do víc instrukcí s pomocnými registry: $f = (g+h) - (i+j)$; se přeloží na `add t0,g,h / add t1,i,j / sub f,t0,t1`.

Podmínka (if). Otestuje se podmínka a *podmíněným skokem* se obejde větev, která se nemá provést. Pohodlně se kóduje skok při *neplatnosti* podmínky (do větve **else**), na konci větve *then* je nepodmíněný skok za **else**:

```
if (a < 3) b = 4; else c = a << 2;
jge    [a], 3, Else // if !(a < 3) jump to Else
store [b], 4        // then-branch: b = 4
jmp    End
Else:
load   r1, [a]       // else-branch: c = a << 2
shl    r1, r1, 2
store [c], r1
End:
```

Tj. test podmínky se obrátí ($a < 3$ versus skok při $a \geq 3$) a podle něj se přeskočí příslušná část kódu.

Cyklus (for). Cyklus rozložíme na: *inicializaci*, *test podmínky*, *tělo*, *aktualizaci* řídicí proměnné a *zpětný skok*. Obvyklé schéma „test na začátku“ skočí hned po inicializaci na podmínku (umístěnou na konci) a zpětně se na tělo vrací jen tehdy, je-li podmínka splněna – ušetří jeden skok za iteraci:

```
for (int i = 0; i < N; ++i) A[i] = 42;
li    r1, 0        // i = 0
la    r0, A        // r0 = base address of A
j     Cond         // jump to the test first
Body:
sll   r2, r1, 2     // offset = i * 4 (4 = sizeof(int))
add   r2, r2, r0    // address of A[i] = A + i*4
store [r2], 42      // A[i] = 42
addi  r1, r1, 1     // ++i
Cond:
slt   r3, r1, [N]   // r3 = (i < N) ? 1 : 0
bne   r3, 0, Body  // if i < N, repeat the body
```

Klíčové je, že index i se musí vynásobit *velikostí prvku* (`sizeof(A[0])`, zde 4 B), aby vznikla správná bajtová adresa $A + i*4$; posun `sll r2,r1,2` je právě toto násobení čtyřmi.

Volání funkce. Volání musí: předat *parametry*, uložit *návratovou adresu* a předat řízení (`call/jal`), uvnitř funkce si zařídit pracovní prostor, vrátit *návratovou hodnotu* a vrátit řízení (`ret/jr`). Kde přesně leží parametry a návratová hodnota, určuje *konvence volání*:

```
int f(int p) { return p + 1; }
void g() { auto r = f(42); }

f:                                // callee
    addi v0, a0, 1                // return value = p + 1 (p came in a0)
    jr   ra                       // return to caller (address saved in ra)

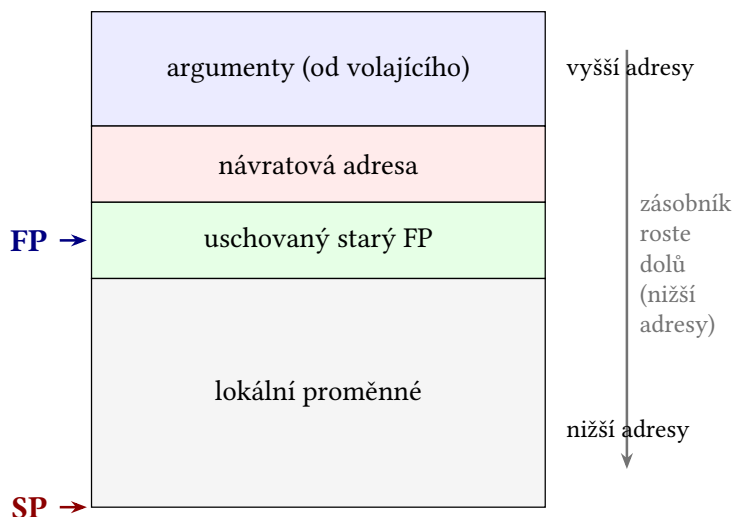
g:                                // caller
    li   a0, 42                   // 1st argument p = 42
    jal  f                       // call: ra <- return address, jump to f
    move r, v0                    // r = return value (in v0)
```

Registr `a0` nese první argument, `v0` návratovou hodnotu, `ra` návratovou adresu. Instrukce `jal f` (jump-and-link) uloží do `ra` adresu *následující* instrukce a skočí na `f`; `jr ra` se vrátí zpět.

📖 Definice — ABI a konvence volání

ABI (*Application Binary Interface*) je dodatek k ISA, který stanovuje, *jak se procesor používá*: význam jednotlivých registrů, rozvržení zásobníku, a *konvenci volání* (calling convention) – kde se předávají argumenty (v určených registrech, přebytné na zásobníku), kde návratová hodnota, kdo (volající/volaný) uschovává které registry. Díky ABI mohou spolu fungovat odděleně přeložené části programu (program a knihovna).

Když je parametrů víc, než kolik se vejde do registrů, nebo při rekurzi, ukládají se hodnoty na *zásobník* (stack): souvislou oblast paměti rostoucí směrem k *nižším* adresám, s ukazatelem vrcholu *SP* (stack pointer). Každé aktivní volání má na zásobníku svůj *rámec* (stack frame / activation record): argumenty, návratovou adresu, uschovaný starý ukazatel rámce a lokální proměnné. Začátek rámce drží *FP* (frame/base pointer), aby se k argumentům a lokálům dalo přistupovat pevnými offsety.



Zásobníkový rámec volané funkce (zásobník roste dolů, vyšší adresy nahoře). Instrukce `call` (x86) uloží na zásobník *návratovou adresu* (kam se po funkci pokračuje) a předá řízení; na MIPS ji `jal` uloží do `ra` a na zásobník ji uschová až prolog funkce, která sama volá další funkce. *Prolog* funkce pak uschová *starý FP* a nastaví nový FP na začátek svého rámce; pod ním si SP odsune místo pro lokální proměnné. K argumentům i lokálům se přistupuje pevnými offsety vůči FP. *Epilog* obnoví starý FP a `ret` se vrátí na uloženou návratovou adresu.

2.4 RISC vs. CISC

i Poznámka. Instrukční sady se hrubě dělí na *RISC* (*Reduced Instruction Set Computer* – málo jednoduchých, stejně dlouhých instrukcí, aritmetika jen mezi registry, přístup do paměti jen přes load/store; např. MIPS, ARM, RISC-V) a *CISC* (*Complex Instruction Set Computer* – mnoho instrukcí proměnné délky, operace i přímo nad pamětí; např. x86). RISC kód je jednodušší na dekódování a zřetězení (pipelining), CISC kód bývá kompaktnější. Rozdíl se v praxi stírá: dnešní x86 procesory uvnitř překládají složité instrukce na jednoduché mikrooperace.

2.5 Řešené příklady

V sadě reálných SZZ otázek tohoto okruhu žádná čistě „instrukční“ úloha není; následující dva krátké příklady proto slouží jen k procvičení obou směrů z požadavků (jazyk ↔ instrukce).

Příklad – Přeložení podmínky do instrukcí

Zadání. Přeložte do symbolických instrukcí konstrukci

```
if (a < 3) b = 4; else c = a << 2;
```

a okomentujte jednotlivé instrukce.

Řešení. Test podmínky obrátíme a podmíněným skokem přeskočíme větev then do **else**; konec then ukončíme nepodmíněným skokem za celou podmínku. Posun <<2 je jediná instrukce (násobení 4):

```
jge    [a], 3, Else    // if !(a < 3) -> Else (negated test)
store  [b], 4          // then: b = 4
jmp    End
Else:
load   r1, [a]          // else: load a
shl    r1, r1, 2         // a << 2
store  [c], r1          // c = a << 2
End:
```

Výsledek: test + podmíněný skok + dvě větve + nepodmíněný skok. Kdyby ISA neměla přímý test „větší rovno“, nahradíme `jge [a], 3` dvojicí `slt r0, [a], 3` (nastaví `r0=1` při `a<3`) a `beq r0, 0, Else` (skok, když `r0=0`, tj. když `a>=3`).

Příklad – Rekonstrukce konstrukce z instrukcí

Zadání. Dána je sekvence instrukcí (registr `r0` drží bázovou adresu pole `A`, `N` je proměnná, prvek pole má 4 bajty). Určete, jaké konstrukci vyššího jazyka odpovídá, a zapište ji zpět v C.

```
li     r1, 0            // r1 = 0
j      Cond
Body:
sll    r2, r1, 2         // r2 = r1 * 4
add    r2, r2, r0        // r2 = A + r1*4
store  [r2], r1          // M[r2] = r1
addi   r1, r1, 1         // r1 = r1 + 1
Cond:
slt    r3, r1, [N]       // r3 = (r1 < N)
bne    r3, 0, Body      // if r1 < N -> Body
```

Řešení. Čteme strukturu: `r1` se inicializuje na 0, na konci těla se o 1 zvyšuje (`addi r1, r1, 1`), a test `slt r3, r1, [N]` / `bne r3, 0, Body` tvoří podmínku `r1 < N` na konci se zpětným skokem – to je *počítaný cyklus* s řídicí proměnnou `r1`. V těle se `sll r2, r1, 2` / `add r2, r2, r0` počítá adresa `A + r1*4`, tedy `&A[r1]`,

a `store [r2], r1` tam zapíše hodnotu `r1`. Vynásobení čtyřmi je právě `sizeof(int)`. Zpětně tedy:

```
for (int i = 0; i < N; ++i) A[i] = i;
```

kde `r1` hraje roli `i`. (Skok na `cond` hned po inicializaci je obvyklé schéma „test na začátku“: tělo se neprovede vůbec, je-li `N <= 0`.)

3 Podpora běhu operačního systému

Operační systém (OS) je program, který stojí mezi aplikacemi a „holým“ hardwarem. Tato sekce vysvětluje, na čem stojí jeho schopnost tuto roli bezpečně zastat: na *dvou režimech procesoru* (uživatelský a privilegovaný), na řízeném přechodu mezi nimi přes *systémové volání* a na *jádře* OS, které v privilegovaném režimu spravuje prostředky. Detailní mechanismy obsluhy zařízení, procesů a paměti rozebírají následující sekce; zde jde o ten základní rámec.

3.1 Role operačního systému

Definice – Operační systém: dvě role

Abstraktní (rozšířený) stroj: OS skrývá složitost a rozmanitost hardwaru a nabízí aplikacím jednotné, na konkrétním HW nezávislé rozhraní – *API jádra* v podobě *systémových volání* (typicky obalených do knihoven jazyka). Místo sektorů na disku či rámců na síťové kartě vidí aplikace soubory, streamy a sokety. *Správce prostředků* (resource manager): veškerý hardware spravuje OS a sdílí jej mezi aplikacemi – *alokuje* paměť, *sdílí v čase* (time-sharing) procesor a *abstrahuje* zařízení (disk, síť).

 **Intuice.** Bez OS by každá aplikace musela umět ovládat všechna zařízení všech výrobců, hlídat, aby si vzájemně nepřepsaly paměť, a explicitně se domlouvat na sdílení procesoru. OS dá každé aplikaci *iluzi*, že má paměť, procesor i zařízení jen pro sebe. Aby tuto iluzi mohl udržet a vynutit (žádná aplikace nesmí sahat na HW přímo), potřebuje *hardwarovou podporu* – právě dva režimy procesoru.

3.2 Privilegovaný a neprivilegovaný režim procesoru

Definice – Uživatelský a privilegovaný režim

Procesor pracuje v jednom ze dvou základních režimů. *Uživatelský režim* (user mode) je neprivilegovaný: běží v něm aplikace. Část prostředků je nedostupná nebo jen pro čtení – zejména *systémové (řídící) registry* a *privilegované instrukce* (in/out, instrukce měnící režim, obsluhu přerušení a tabulky překladu adres). Pokus o privilegovanou instrukci vyvolá výjimku. Paměť jádra a paměť mapovanou na zařízení aplikace nevidí, protože ji OS nenamapuje do jejího adresového prostoru (překlad adres, sekce 6). *Privilegovaný (jádrový, systémový) režim* (kernel mode) je plně oprávněný: jsou dostupné *všechny* registry a instrukce a je v něm *plný přístup ke všem prostředkům*. Běží v něm jádro OS, případně jen jeho část.

 **Poznámka.** Architektura x86 zavádí čtyři úrovně oprávnění zvané *ringy* (ring 0 až ring 3). V praxi se používají jen dvě krajní: *ring 3* odpovídá uživatelskému režimu (aplikace) a *ring 0* privilegovanému režimu (jádro). Prostřední ringy 1 a 2 běžné OS nevyužívají. Aktuální úroveň oprávnění drží procesor ve svém stavu a kontroluje ji u každé citlivé instrukce.

3.3 Přejít mezi režimy a systémové volání

Aplikace nesmí volat libovolný kód jádra: nezná jeho adresy a hlavně by tím obešla ochranu. Přejít do privilegovaného režimu jde proto *jen* přes pevně daný *vstupní bod jádra* (entry point).

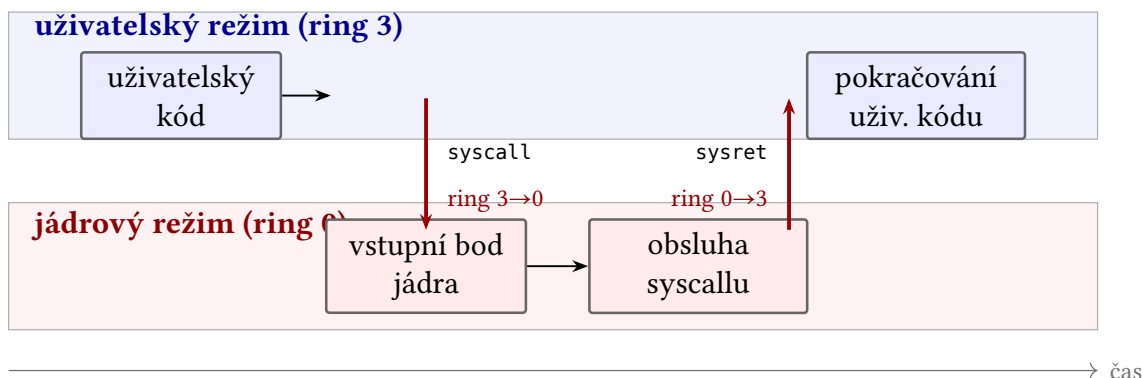
Definice – Systémové volání (syscall)

Systémové volání je požadavek aplikace na službu jádra. Uživatelská instrukce `syscall` (na starších x86 přerušení `int 0x80`) způsobí, že procesor *přepne do privilegovaného režimu* a skočí na adresu vstupního bodu jádra uloženou v systémovém registru (na x86-64 v registru `IA32_LSTAR`). Číslo služby a argumenty předá aplikace v registrech. Jádro požadavek obslouží a instrukcí `sysret` se vrátí zpět do uživatelského režimu na následující instrukci aplikace.

 **Intuice.** Klíčové je, že cílovou adresu skoku určuje *jádro* (uložilo si ji při startu), ne aplikace. Aplikace tedy nemůže „skočit doprostřed“ jádra s právy ring 0; smí jen na definovaný vstupní bod, kde si jádro zkontroluje, co po něm chce. To je celé jádro ochrany: přechod oprávnění je svázán s jediným pevným vstupem.

I zdánlivě nevinný řádek vyššího jazyka se obvykle opírá o systémové volání. Třeba výpis na obrazovku vede přes knihovnu (`printf`, která text nejdříve ukládá do svého bufferu) na systémové volání zápisu (`write`), které data předá jádru, protože obrazovka i standardní výstup jsou *sdílené prostředky* spravované OS:

```
// User mode (ring 3): the application just calls a library function
int main() {
    printf("Hello"); // wrapped in libc -> issues a write() syscall
}
// The write() wrapper sets up registers and executes one instruction:
// syscall jump to kernel entry, switch to ring 0
// ... kernel handles the write to standard output
// sysret return to ring 3, continue after the syscall
```



Systémové volání jako přechod mezi režimy. Uživatelský kód běží v horní dráze (ring 3). Instrukce `syscall` přepne procesor do privilegovaného režimu a předá řízení *jedinému pevnému vstupnímu bodu jádra* (skok dolů), nikam jinam – to je podstata ochrany. Jádro v dolní dráze (ring 0) požadavek obslouží a instrukcí `sysret` přepne zpět do ring 3 a pokračuje uživatelský kód za voláním.

3.4 Jádro operačního systému

Definice – Jádro (kernel)

Jádro je ústřední, vždy zavedená část OS, která běží v privilegovaném režimu a poskytuje aplikacím rozhraní přes systémová volání. Spravuje prostředky počítače – typicky obsahuje subsystemy pro *správu paměti, správu a plánování procesů a vláken, souborový systém, síťový subsystem* a *ovladače zařízení*. Jako jediné má přímý přístup k hardwaru.

Podle toho, jak je jádro vnitřně uspořádané a kolik kódu běží v privilegovaném režimu, rozlišujeme základní architektury jádra:

- **Monolitické jádro** – celé jádro je jeden program, subsystémy se volají jako běžné funkce a mají stejná (plná) oprávnění. *Výhody:* nízká režie komunikace, efektivní kód. *Nevýhody:* chyba v kterékoli části může shodit celý systém, žádná izolace subsystémů. Moderní monolity umějí navíc *dynamicky zavádět moduly* (flexibilita za cenu ještě většího rizika). Příklad: **Linux** (monolit s moduly), Solaris, tradiční Unixy.
- **Vrstvené jádro** – subsystémy uspořádané do hierarchie vrstev, vrstva $n+1$ používá výhradně služby vrstvy n . *Výhody:* přehlednější, snáz rozšiřitelné. *Nevýhody:* ne vždy jdou funkce čistě seřadit do vrstev, vrstvy přidávají režii.
- **Mikrojádro** – v privilegovaném režimu zůstane jen nezbytné minimum (obsluha přerušení, komunikace mezi procesy); subsystémy (souborový systém, ovladače, ...) běží jako oddělené *servery* v uživatelském režimu a komunikují předáváním *zpráv* (klient/server). *Výhody:* bezpečnost, robustnost, snadná rozšiřitelnost (chyba serveru neshodí jádro). *Nevýhody:* pomalejší kvůli režii zpráv a častému přepínání režimu. Příklady: MINIX, QNX, L4; myšlenka je částečně přejatá i jinde (návrh jádra Windows, Linux FUSE – souborový systém v user space).

i Poznámka. Volba architektury je *kompromis mezi izolací (odolností) a efektivitou*: izolované subsystémy jsou bezpečnější, ale komunikace mezi nimi přes hranici režimu stojí čas. Podrobnosti jednotlivých architektur už jdou nad rámec doslovných požadavků okruhu (ty chtějí „jádro operačního systému“), proto je držíme heslovitě.

3.5 Řešený příklad: které operace vyžadují jádrový režim

Příklad – Které operace potřebují přechod do jádra

Zadání.

U každé z následujících operací rozhodněte, zda vyžaduje přechod do jádrového režimu (systémové volání), nebo proběhne celá v uživatelském režimu: (a) čtení souboru z disku, (b) sečtení hodnot dvou registrů, (c) vyžádání nové paměti od OS (např. `malloc`, který nemá dost ve své rezervě), (d) zápis textu na obrazovku, (e) inkrementace lokální proměnné v cyklu.

Řešení.

Pravidlo: *přechod do jádra je nutný právě tehdy, sahá-li operace na sdílený hardwarový prostředek, který spravuje OS* (disk, paměťové mapování procesu, obrazovka/terminál, síť). Čistý výpočet nad registry a vlastní paměti procesu žádné syscall nepotřebuje.

- (a) **Čtení souboru** – *ano, syscall* (`read/open`). Disk je sdílený prostředek, přístup k němu jde jedine přes jádro a jeho ovladač.
- (b) **Sečtení dvou registrů** – *ne*. Jedna aritmetická instrukce, čistý výpočet v user mode; nesáhá na žádný sdílený prostředek.
- (c) **Alokace nové paměti od OS** – *ano, syscall* (`mmap/brk`). Rozšířit adresový prostor procesu (zařídí mapování stránek) umí jen jádro. *Pozn.:* pokud `malloc` vystačí s už dříve získaným blokem, žádný syscall se nekoná – ten je čistě v user space.
- (d) **Zápis na obrazovku** – *ano, syscall* (`write`). Standardní výstup/terminál je sdílený prostředek spravovaný OS (viz `printf` výše).
- (e) **Inkrement lokální proměnné** – *ne*. Lokální proměnná žije v registru nebo na zásobníku procesu; inkrement je jedna instrukce, žádné sdílení.

Vyžadují jádro tedy (a), (c), (d); čistě v user módu proběhnou (b) a (e).

4 Rozhraní periferií a jejich obsluha

Periferní zařízení (disk, myš, síťová karta) jsou výrazně pomalejší a rozmanitější než procesor a paměť, a přitom s nimi procesor musí umět komunikovat jednotně. Tato sekce vysvětluje, jak je zařízení připojeno (přes *řadič*), jak s ním procesor mluví (I/O porty vs. paměťově mapované registry), jak vypadá *programem řízená obsluha* s aktivním čekáním a jak funguje alternativa – *přerušování* – včetně přesné reakce hardwaru a operačního systému na žádost o přerušování. To přímo pokrývá oba doslovné požadavky okruhu: implementovat obsluhu zadaného zařízení pro dané porty a popsat reakci na přerušování na úrovni instrukcí.

4.1 Zařízení, řadič a ovladač

Definice – Zařízení, řadič, ovladač

Zařízení (device) je hardwarová komponenta určená pro konkrétní účel (disk, myš, klávesnice, síťová karta). *Řadič zařízení* (device controller) je hardware, který zařízení elektricky obsluhuje – pracuje se signály „na drátech“, převádí mezi analogovým a digitálním světem a spravuje jedno či více zařízení připojených v nějaké topologii (sběrnice, hvězda, strom). Navenek vystavuje sadu *registrů*, přes které procesor zadává příkazy a čte stav. *Ovladač zařízení* (device driver) je naopak *softwarová* komponenta – část jádra OS (typicky modul, často zaváděný dynamicky) – která řadič ovládá zápisem/čtením jeho registrů a vyšší vrstvě OS nabízí *abstraktní rozhraní* (např. „přečti blok“, „pošli paket“). Ovladač je specifický pro konkrétní řadič nebo třídu řadičů.

 **Intuice.** Řadič je „hardwarová strana“ a ovladač „softwarová strana“ téhož rozhraní. Procesor sám neví, jak roztočit ploténku disku nebo přečíst pozici myši; ví jen, jak číst a zapisovat čísla do registrů řadiče. Řadič je tlumočnick mezi tímto jednoduchým registrovým rozhraním a fyzikou konkrétního zařízení. Ovladač tuto registrovou „abecedu“ zná a překládá ji na pojmy, kterým rozumí zbytek jádra. Díky této dvojici vrstev může zbytek OS pracovat se všemi disky stejně, bez ohledu na výrobce.

 **Poznámka.** Při startu počítače se zařízení *vyjmenují a inicializují* přes základní HW rozhraní BIOS/UEFI (firmware na základní desce): zjistí se, která zařízení jsou připojena a na jakých adresách jsou jejich registry, a předá se řízení zaváděcí OS. Teprve potom převezme zařízení do správy jádro se svými ovladači.

4.2 Jak procesor komunikuje se zařízením

Procesor čte a zapisuje registry řadiče jedním ze dvou způsobů:

- **Speciální I/O instrukce.** Procesor má zvláštní (privilegované) instrukce pro vstup a výstup, které adresují zařízení *číslem portu* v odděleném *I/O adresovém prostoru*. Na x86 jsou to instrukce `in` a `out` (např. `in al, dx` přečte bajt z portu, jehož číslo je v registru `dx`). Port je tedy „adresa registru řadiče“ v tomto zvláštním prostoru.
- **Paměťově mapovaná zařízení** (memory-mapped I/O). Registry řadiče jsou namapovány na pevné adresy v *běžném* adresovém prostoru. Procesor pak se zařízením komunikuje obvyčejnými instrukcemi `load/store` (čtení a zápis do paměti), které hardware podle adresy přesměruje na řadič místo do RAM.

Definice – Paměťově mapované I/O

Paměťově mapované I/O přiřazuje řídicím a stavovým registrům zařízení pevné adresy v adresovém prostoru procesoru. Běžné čtení (`load`) z takové adresy hardware přeloží na čtení registru zařízení a běžný zápis (`store`) na zápis do registru; zápis na určitý *offset* přitom typicky funguje jako *příkaz* pro zařízení (např. zápis čísla do registru *Command* spustí čtení sektoru). Pro program tak ovládání zařízení vypadá jako práce s pamětí, jen na vyhrazených adresách.

💡 Intuice. U paměťově mapovaného I/O je klíčové, že *zápis sám o sobě něco udělá* – nejde o uložení hodnoty „na později“, ale o povel. Proto se takové registry v jazyce C deklarují jako **volatile** (viz též rozbor u řešeného příkladu níže): překladač nesmí čtení/zápisy slučovat, přeskakovat ani přehazovat, i když se mu „opakované čtení téže adresy“ zdá zbytečné. Pro každé čtení stavového registru totiž může řadič vrátit jinou hodnotu, aniž by do něj cokoli v programu zapsalo.

4.3 Programem řízená obsluha (PIO) s aktivním čekáním

Definice – Programem řízená obsluha (PIO), aktivní čekání

Programem řízená obsluha (Programmed I/O, PIO) je způsob obsluhy zařízení, při kterém *celou* operaci řídí instrukce procesoru čtením a zápisem registrů řadiče. Řadič vystavuje *řídící* registry (kam procesor zapisuje příkazy a parametry) a *stavové* registry (odkud procesor čte aktuální stav, typicky příznaky jako „zanepřázdněn“ a „chyba“). Při *aktivním čekání* (polling, busy-waiting) procesor po zadání příkazu v cyklu opakovaně čte stavový registr, dokud zařízení operaci nedokončí.

Typický průběh čtení sektoru z disku v PIO s aktivním čekáním (pro daný řadič s danými offsety/funkcemi portů) má tyto kroky:

- (1) **Zkontroluj stav.** Přečti stavový/chybový registr a ujisti se, že řadič je připraven (není zaneprázdněn, nehlásí chybu).
- (2) **Nastav parametry.** Zapiš do parametrových registrů, co a kam: číslo bloku (LBA, případně sektor/cylindr/hlava), počet bloků, cílovou adresu v paměti.
- (3) **Pošli příkaz.** Zápisem do *příkazového* registru (např. kódu „čtení“) spust operaci. Tím se zařízení rozběhne a nastaví příznak „zanepřázdněn“.
- (4) **Čekej na dokončení.** Opakovaně čti stavový registr, dokud nezmizí příznak „zanepřázdněn“ (nebo dokud se neobjeví příznak „chyba“). To je ono aktivní čekání.
- (5) **Přeber data.** Po dokončení přečti data (u jednoduchých řadičů z datového registru, u moderních je už přenesl do paměti řadič sám, viz DMA níže) a zkontroluj, zda nenastala chyba.

Stejný princip (cyklické čtení stavového a datového registru) platí pro libovolné zařízení obsluhované PIO. Třeba u *myši* program čeká, až stavový registr ohlásí nová data, a pak z datového registru přečte relativní pohyb a stav tlačítek; liší se jen význam jednotlivých registrů a portů.

💡 Intuice. Aktivní čekání je jednoduché, ale plýtvavé: procesor v kroku (4) jen dokola čte tutéž adresu a nic užitečného nedělá, zatímco pomalé zařízení pracuje. Pro krátké operace nebo v situaci, kdy stejně nemáme co počítat (např. inicializace ovladače při bootu), je to v pořádku. Pro běžný provoz se polling nahrazuje *přerušením* – procesor zadá příkaz a věnuje se jiné práci, dokud ho zařízení samo neupozorní, že je hotovo.

4.4 Přerušení

Definice – Přerušení (interrupt)

Přerušení je mechanismus, kterým je tok instrukcí procesoru *asynchronně* přesměrován na zvláštní *obslužnou rutinu* (interrupt handler). Místo aby procesor stav zařízení sám zjišťoval pollingem, zařízení samo signalizuje procesoru, že vyžaduje pozornost – typicky vystavením signálu na vyhrazeném vstupu procesoru (*IRQ pin*, žádost o přerušení). Procesor po dokončení (nebo zrušení) právě prováděné instrukce přeruší běžný tok, vykoná obsluhu a poté se vrátí tam, kde byl přerušen.

Přerušení se podle zdroje dělí na tři typy:

- **Externí (hardwarové) přerušení** – pochází od zařízení přes IRQ pin (dokončení diskové operace, příchod paketu, stisk klávesy, tik časovače). Tato přerušení lze dočasně *maskovat* (zakázat), když jádro provádí citlivou operaci, která nesmí být přerušena.

- **Výjimka** (exception) – vyvolá ji *instrukce* sama během svého provádění. Architektura má pevně daný seznam výjimek. *Trap* (past) se vyvolá *po* instrukci (např. krokování při ladění), *fault* (chyba) *před* dokončením, instrukce se zruší (rollback) a po obsluze se může zopakovat (typicky výpadek stránky – page fault, dále dělení nulou, neplatný opkód, výpadek ochrany).
- **Softwarové přerušení** – vyvolá je speciální instrukce (na x86 *int*). Hodí se mj. jako mechanismus *systémového volání* (starší x86 používaly *int 0x80*, viz sekce 3).

i Poznámka. Na architektuře x86 je celkem **256** přerušení očíslovaných 0 až 255; *prvních 32* je vyhrazeno architekturou pro výjimky (např. 0 = dělení nulou, 0x0E = výpadek stránky, 0x0D = výpadek ochrany), zbytek slouží pro IRQ od zařízení a softwarová přerušení. Číslo přerušení slouží jako index do *tabulky vektorů přerušení* (na x86 tzv. IDT), která pro každé číslo udává adresu příslušné obslužné rutiny.

4.5 Reakce na žádost o přerušení (hardware + software)

Toto je jádro druhého požadavku okruhu: na úrovni vykonávání instrukcí popsat, co dělá *procesor* (hardware) a co *jádro OS* (software), když přijde žádost o přerušení. Reakce probíhá v následujících krocích; je užitečné je rozdělit podle toho, kdo je provádí.

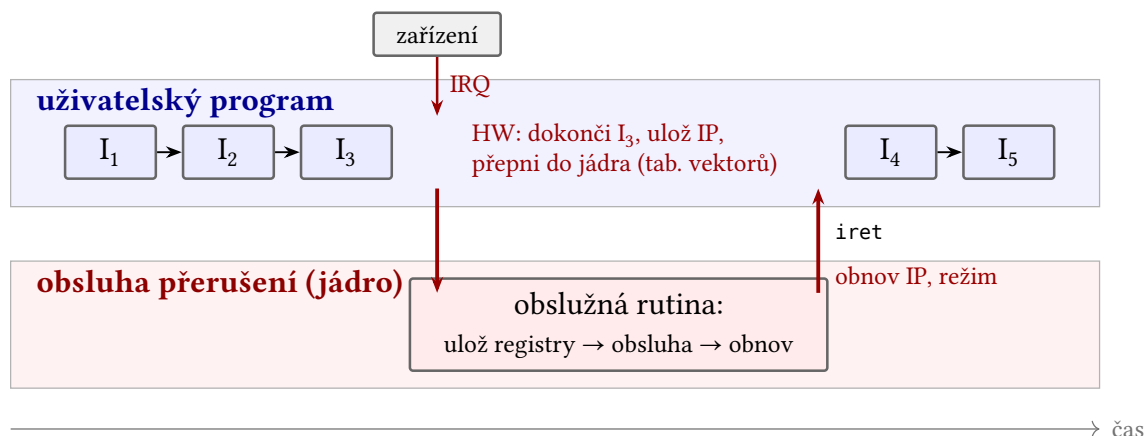
Hardware (procesor) automaticky:

- (1) **Dokončí (nebo zruší) aktuální instrukci.** Přerušení se zpracuje *mezi* instrukcemi – právě běžící instrukce se buď doběhne (u externích přerušení, traps), nebo se zruší a stav se vrátí před ni (u faultů, aby ji šlo po obsluze zopakovat). Procesor tak nikdy nezůstane v „půlce“ instrukce.
- (2) **Zjistí zdroj přerušení.** Z čísla přerušení (předaného IRQ radičem nebo daného typem výjimky) určí, o jaké přerušení jde.
- (3) **Najde adresu obsluhy.** Číslem přerušení indexuje *tabulku vektorů přerušení* (pole ukazatelů na obslužné rutiny) a získá adresu příslušné rutiny.
- (4) **Uloží kritickou část stavu a přepne oprávnění.** Uloží aspoň ukazatel instrukce *IP* (a příznaky), aby se měl kam vrátit, a přepne procesor do *privilegovaného režimu* – obslužná rutina je součástí jádra.
- (5) **Skočí na obsluhu.** Nastaví *IP* na adresu z kroku (3); od této chvíle běží kód obslužné rutiny.

Software (obslužná rutina jádra) pak:

- (6) **Uloží zbytek stavu.** Rutina uschová (typicky na zásobník) ostatní registry, které bude přepisovat, aby je na konci mohla obnovit netknuté.
- (7) **Vykoná vlastní obsluhu.** Udělá, co dané přerušení vyžaduje: přebere data ze zařízení / potvrdí dokončení operace, u výpadku stránky doplní mapování, u softwarového přerušení obslouží systémové volání atd.
- (8) **Obnoví uložený stav** z kroku (6).
- (9) **Vrátí se** instrukcí návratu z přerušení (na x86 *iret*). Ta obnoví kritický stav z kroku (4) – *IP* a příznaky – a přepne zpět do původního (uživatelského) režimu.

Procesor pak pokračuje v přerušeném toku instrukcí přesně tam, kde byl, jako by se nic nestalo (u faultu se zopakuje zrušená instrukce). Dělicí čára je tedy: *hardware* zajistí přepnutí kontextu na obsluhu (uloží *IP*, přepne režim, skočí přes tabulku vektorů) a přepnutí zpět; *software* (obslužná rutina) odvede vlastní práci a uschová/obnoví zbytek registrů.



Časová osa obsluhy přerušení. Uživatelský program vykonává instrukce $I_1, I_2, \dots$ (horní dráha). Mezi I_3 a I_4 vystaví zařízení žádost IRQ. **Hardwarová** reakce (procesor): dokončí I_3 , uloží IP, přepne do jádra a přes tabulku vektorů přerušení skočí na obslužnou rutinu (dolní dráha). **Softwarová** reakce (rutina v jádře): uloží registry, provede vlastní obsluhu, obnoví registry a instrukcí `iret` se vrátí (obnova IP a režimu). Program pak pokračuje instrukcí I_4 , jako by se nic nestalo.

i Poznámka. **DMA** (Direct Memory Access, přímý přístup do paměti) je doplňkový mechanismus přenosu *dat* bez zatěžování procesoru: místo aby data mezi zařízením a pamětí kopíroval po slovech procesor, zadá řadiči (resp. DMA řadiči) cílovou fyzickou adresu a ten přenesení celý blok sám. Procesor se mezitím věnuje jiné práci a o dokončení přenosu se dozví přerušením. DMA tak typicky doplňuje obsluhu přerušením; v následujícím příkladu disk přenáší přečtený blok do paměti právě přes DMA, takže ovladač jen nastaví cílovou adresu a čeká na dokončení.

4.6 Řešený příklad: ovladač pro řadič disku

Příklad — SZZ jaro 2024 č. 2: Ovladač pro řadič disku

(Reálná otázka SZZ: jaro 2024, č. 2 – Ovladač pro řadič disku)

Zadání.

Napište implementaci funkce ovladače disku pro načtení jednoho bloku. Disk je řízen pěti *paměťově mapovanými* registry (zápis příkazů a parametrů, čtení stavu); zařízení nepodporuje přerušování, obsluha je s aktivním čekáním. Registry (offsety v bajtech od základní adresy):

Offset	Registr	Typ	Popis
0	Status	R	Bitové pole aktuálního stavu řadiče (zápisy ignorovány).
4	Size	R	Velikost disku v blocích (zápisy ignorovány).
8	Command	W	Zápis spustí příkaz (1 = čtení, 2 = zápis).
12	LBA	W	Logická adresa bloku pro příkaz v <i>Command</i> .
16	DMA	W	Fyzická adresa paměti pro uložení čtených dat.

Stavový registr (offset 0) je bitové pole; bit 0 = **ERR** (1 = chyba), bit 1 = **BUSY** (nastaven po zadání příkazu, vynulován po dokončení operace), ostatní bity nuly. Blok má 512 B, přenos dat zařizuje DMA (stačí nastavit fyzickou cílovou adresu). Systém je 32-bitový, disk max. 2 TB. Doplňte těla funkcí a navrhnete strukturu `disk_t` pro uchování informací o disku (k systému může být připojeno více disků, rozlišených různými instancemi). Obě funkce vracejí `true` při úspěchu, jinak `false`. `disk_init` se volá jednou při inicializaci (parametr `register_address` je virtuální adresa mapovaných registrů, tj. adresa stavového registru). `disk_read_block_waiting` přečte jeden blok na fyzickou adresu `data_phys_addr`; návrat až po dokončení (aktivní čekání povoleno). Implementace musí aspoň triviálně ošetřit současný přístup z více procesů.

Řešení.

(1) Struktura dat a mapování registrů. Registry řadiče popíšeme jako `struct`, jejíž členy padnou přesně na dané offsety (pět 32-bitových slov `uint32_t` za sebou: Status@0, Size@4, Command@8, LBA@12, DMA@16). Tuto strukturu deklarujeme `volatile`, protože jde o paměťově mapované registry: čtení stavu musí proběhnout *pokaždé* znovu (řadič hodnotu mění „zvenčí“) a zápisy se nesmějí slučovat ani přehazovat – jsou to povely. Ovladač si o disku pamatuje ukazatel na registry, velikost disku a *zámek* (mutex):

```
typedef volatile struct {
    uint32_t status;    // offset 0: bit0 = ERR, bit1 = BUSY
    uint32_t size;      // offset 4: disk size in blocks
    uint32_t command;   // offset 8: 1 = read, 2 = write
    uint32_t lba;       // offset 12: block address for the command
    uint32_t dma;       // offset 16: physical target address (DMA)
} disk_regs_t;

typedef struct {
    size_t max_lba;     // disk size (for range checking)
    disk_regs_t *ctl;   // pointer to the memory-mapped registers
    mutex_t mutex;      // serializes access from multiple processes
} disk_t;
```

(2) Čtení stavu a aktivní čekání. Stav řadiče testují dvě pomocné funkce: `disk_is_ok` ověří, že není nastaven příznak ERR (bit 0), `disk_is_ready` že není nastaven příznak BUSY (bit 1). Aktivní čekání pak v cyklu čte stav, dokud řadič není připraven; když se po cestě objeví chyba, čekání skončí neúspěchem:

```
static bool disk_is_ok(disk_t *disk) { return (disk->ctl->status & 1) == 0; }
static bool disk_is_ready(disk_t *disk) { return (disk->ctl->status & 2) == 0; }

static bool disk_wait_for_ready(disk_t *disk) {
    while (!disk_is_ready(disk)) { // busy-wait while BUSY bit is set
        if (!disk_is_ok(disk)) { // ERR bit appeared -> give up
            return false;
        }
    }
    return true;
}
```

Pozn. k závorkování: oficiální náčrt SZZ má v těchto testech `disk->ctl->status & 1 == 0` bez vnitřních závorek. V jazyce C má ale operátor `==` vyšší prioritu než bitové `&`, takže výraz se vyhodnotí jako `status & (1 == 0) = status & 0 = 0`, tj. vždy `false` – to je chyba. Zamýšlené je `(status & 1) == 0`. Závorky proto doplňujeme; jde o klasickou past s prioritou operátorů (bitové operátory mají v C nižší prioritu než relační).

(3) Inicializace. `disk_init` uloží (přetypuje) virtuální adresu registrů do struktury, přečte velikost disku, inicializuje zámek a počká, až je řadič připraven:

```
bool disk_init(disk_t *disk, uint32_t register_address) {
    disk->ctl = (disk_regs_t *) register_address;
    disk->max_lba = disk->ctl->size;
    mutex_init(&disk->mutex);
    return disk_wait_for_ready(disk);
}
```

(4) Čtení bloku s aktivním čekáním. Funkce nejprve ověří rozsah LBA, pak *zamkne* mutex (kvůli více procesům), počká na připravenost řadiče, nastaví parametry (LBA, cílovou DMA adresu) a zápisem do *Command* spustí čtení (kód 1). Poté znovu aktivně čeká na dokončení (zhasnutí BUSY) a zámek uvolní:

```
bool disk_read_block_waiting(disk_t *disk, size_t lba, uint32_t data_phys_addr) {
    if (lba >= disk->max_lba) { // out-of-range block
        return false;
    }
    // ... (rest of the function code) ...
}
```

```

}
mutex_lock(&disk->mutex);           // only one process drives the disk
bool ok = disk_wait_for_ready(disk);
if (!ok) {
    mutex_unlock(&disk->mutex);
    return false;
}
disk->ctl->lba = lba;                // set parameters ...
disk->ctl->dma = data_phys_addr;
disk->ctl->command = 1;              // ... then issue READ -> sets BUSY

ok = disk_wait_for_ready(disk);     // busy-wait until the read completes
mutex_unlock(&disk->mutex);
return ok;
}

```

Rozbor. Proč je řešení takové, jaké je:

- (1) **Proč *volatile*.** Registry jsou paměťově mapované. Bez *volatile* by překladač viděl smyčku, která jen čte tutéž adresu, aniž do ní cokoli zapisuje, a mohl by čtení „vytknout“ mimo cyklus – pak by se BUSY/ERR nikdy nepřčetlo znovu a smyčka by se zacyklila. *volatile* říká: každý přístup proved skutečně a v daném pořadí.
- (2) **Proč zámek (*mutex*).** Řadič má jedinou sadu registrů a zvládne jen jednu operaci naráz. Kdyby dva procesy současně nastavily LBA/DMA a oba zapsaly Command, povely by se proložily a data by skončila zpřeházená (nebo by jedna operace zničila parametry druhé). Mutex obě kritické sekce *serializuje*: druhý proces počká, než první dočte a odemkne. To je ono „aspoň triviální ošetření současného přístupu“. Důležité je odemknout zámek *na všech* návratových cestách (i při chybě), jinak by vznikl deadlock.
- (3) **Jak funguje aktivní čekání na BUSY.** Po zápisu příkazu řadič nastaví bit BUSY a rozběhne se. Smyčka `disk_wait_for_ready` čte stavový registr a točí se, dokud je BUSY nastaven. Procesor přitom „nic užitečného“ nedělá – to je daň za jednoduchost a za to, že zařízení nepodporuje přerušování (jinak by tu bylo blokující čekání místo aktivního).
- (4) **Kontrola ERR.** Uvnitř čekací smyčky se navíc testuje bit ERR: objeví-li se chyba (ERR= 1) ještě *v době, kdy řadič drží BUSY*, čekání skončí a funkce vrátí `false`, aby chyba neuvázla v nekonečném čekání. (Rohový případ, kdy by řadič v témže stavovém slově zhasl BUSY a zároveň rozsvítil ERR, tato jednoduchá smyčka – stejně jako oficiální náčrt – neošetří a vyhodnotí jako úspěch; reálné řadiče ale BUSY drží až do dokončení operace.) DMA přenos dat do paměti při úspěchu zařizuje řadič sám, ovladač jen předal cílovou adresu.

5 Abstrakce OS: procesy, vlákna, plánování a soubory

Operační systém staví nad holým hardwarem dvě klíčové *abstrakce sdílení prostředků*: sdílení *procesoru* (aby se zdálo, že má každá běžící úloha procesor sama pro sebe – procesy a vlákna) a sdílení *úložného prostoru* (pojmenované soubory). Tato sekce zavádí procesy, vlákna, kontext a jeho přepínání, plánování a stavy plánovací jednotky a krátce souborovou abstrakci. Vychází z režimů procesoru a jádra ze sekce 3; *sdílení paměti a virtuální paměť* (adresový prostor jako mapování VA→FA, stránkování) řeší samostatně sekce 6.

5.1 Proces a vlákno

Definice – Program, proces, vlákno

Program je *pasivní* sada instrukcí a dat (soubor na disku, výstup překladače). *Proces* je *instance* programu vytvořená OS za běhu: dostane vlastní *adresový prostor* (paměť) a vlastní *prostředky* (otevřené soubory, sokety, synchronizační primitiva). Proces v podstatě poskytuje aplikaci virtuální stroj. *Vlákno* (thread) je *jeden tok instrukcí* (jedna aktivita) vykonávaný procesorem uvnitř procesu; je to *jednotka plánování*. Každý proces má aspoň jedno vlákno, může jich mít víc.

 **Intuice.** Hranice „co je či“ je důležitá. Vlákno má *vlastní* jen to, co je nutné k jeho samostatnému běhu: pozici v kódu (program counter), *zásobník* (stack – lokální proměnné a návratové adresy) a *stav procesoru* (obsah registrů). Všechno ostatní – kód, statická data, halda, otevřené soubory – vlákna procesu *sdílejí*. Proto je komunikace mezi vlákny levná (sdílená paměť), ale vyžaduje synchronizaci (sekce 7), zatímco procesy jsou navzájem izolované.

Definice – Neprivilegovaný (uživatelský) proces

Neprivilegovaný (uživatelský) proces běží v *uživatelském režimu* procesoru (sekce 3): nemá přímý přístup k hardwaru ani k paměti jádra a ostatních procesů a o privilegované operace musí žádat jádro *systémovým voláním* (syscall). Adresové prostory procesů jsou navzájem oddělené, takže chyba či zlovolný kód jednoho procesu nemůže přímo poškodit ostatní. Tato *izolace* je základní bezpečnostní a spolehlivostní vlastnost OS.

5.2 Kontext a přepínání kontextu

Definice – Kontext plánovací jednotky

Kontext je stav plánovací jednotky (vlákna, resp. procesu), který OS musí uchovat, aby šel její běh později přesně obnovit. Patří do něj:

- *obsah registrů procesoru* – včetně program counteru (IP/PC), stack pointeru (SP), stavového slova (příznaky), případně vektorových a FP registrů;
- *stav virtuální paměti* – ukazatel na stránkovací tabulky a obsah TLB (podrobně sekce 6); ten se přepíná při změně *procesu*, ne mezi vlákny téhož procesu, která adresový prostor sdílejí.

Cache jsou *transparentní* – nejsou součástí kontextu (jen ovlivní výkon).

Definice – Přepnutí kontextu (context switch)

Přepnutí kontextu je operace, při níž OS *uloží* kontext právě běžící jednotky (do její struktury v jádře) a *obnoví* kontext jiné jednotky, kterou nasadí na procesor. Je to *drahá* operace (řádově stovky až tisíce instrukcí): kromě uložení a obnovy registrů zahrnuje u změny procesu i přepnutí adresového prostoru, což (bez ASID, sekce 6) zneplatní položky v TLB (a znovunaplnění TLB stojí další přístupy do paměti).

5.3 Multitasking: kooperativní vs. preemptivní

Multitasking je souběžný (concurrent) běh více úloh na (případně jediném) procesoru tak, že se o něj střídají. Liší se podle toho, *kdo a kdy* rozhodne o odebrání procesoru běžící jednotce.

Definice – Kooperativní vs. preemptivní multitasking

Při *kooperativním* multitaskingu se běžící jednotka musí procesoru vzdát *sama a dobrovolně* (vrátí řízení OS, typicky voláním *yield* nebo zablokováním na I/O). OS sám běh nepřerušuje, takže jediná nekooperující jednotka může zablokovat celý systém. Při *preemptivním* multitaskingu má každá běžící jednotka přidělené *časové kvantum* (time slice); když je vyčerpá, *hardwarový časovač* vyvolá přerušování, OS dostane řízení a jednotku *preemptuje* (odebere jí procesor, přepne na jinou). Preemptce tak nevyžaduje spolupráci úloh a zaručuje spravedlivé sdílení – proto ji používají moderní OS pro obecné výpočty.

i Poznámka. Pokud se jednotka v preemptivním režimu zablokuje nebo skončí ještě před vypršením kvanta, časovač není potřeba – OS přepne hned. Kooperativní model se dnes vyskytuje spíše uvnitř běhových prostředí (např. uživatelská vlákna / *fibery* a asynchronní úlohy s explicitním *await*).

5.4 Plánování (scheduling)

Plánovač (scheduler) je část OS, která pomocí *plánovacích algoritmů* rozhoduje, která připravená jednotka poběží na procesoru (jádrě). Volí kompromis mezi několika cíli:

Definice – Cíle plánování

Využití procesoru (utilization) – udržet procesor co nejvíc vytížený. *Propustnost* (throughput) – počet úloh dokončených za jednotku času. *Obrátková doba* (turnaround) – celkový čas od příchodu úlohy do jejího dokončení. *Čekací doba* (waiting time) – čas strávený ve stavu *Ready* (čekáním na procesor). *Doba odezvy* (response time) – čas do první reakce u interaktivních aplikací. K tomu *spravedlnost* (fairness). Cíle jsou často protichůdné (např. krátká odezva vs. vysoká propustnost).

Definice – Základní plánovací algoritmy

FCFS (First Come, First Served) – jediná FIFO fronta; jednotky běží v pořadí příchodu a *bez přerušování* až do konce (nepreemptivní). Jednoduché, ale dlouhá úloha vepředu zdrží všechny za sebou (konvojový efekt).

SJF (Shortest Job First) – přednost dostane úloha s *nejkratší* dobou běhu; při současném příchodu úloh minimalizuje průměrnou čekací dobu, ale vyžaduje znát (či odhadnout) délku úloh předem.

Round Robin (RR) – jako FCFS, ale *preemptivní*: každá jednotka dostane časové kvantum; když ho vyčerpá (nebo se zablokuje), zařadí se na *konec* fronty a procesor dostane další. Malé kvantum dává krátkou odezvu, ale víc přepnutí kontextu (režie).

Víceúrovňová fronta se zpětnou vazbou (multilevel feedback queue) – několik front s rostoucím kvantem; jednotka, která vyčerpá celé kvantum, klesne do nižší (méně časté) fronty, jednotka, která se zablokuje dříve (interaktivní, I/O-vázaná), stoupne do vyšší. Plánuje se vždy hlava nejvyšší neprázdné fronty. Tak se interaktivní úlohy zvýhodní bez znalosti jejich délky předem.

Definice – Priorita

Priorita je číslo vyjadřující důležitost jednotky; jednotka s vyšší prioritou se plánuje dříve (nebo častěji). Typicky jde o součet *statické priority* (přidělené při startu podle důležitosti uživatele/úlohy) a *dynamické priority*, která přidává spravedlnost: OS ji čas od času zvyšuje všem čekajícím (*Ready*) jednotkám a nuluje ji té, která byla právě naplánována – aby dlouho čekající úloha nezůstala „vyhladovělá“.

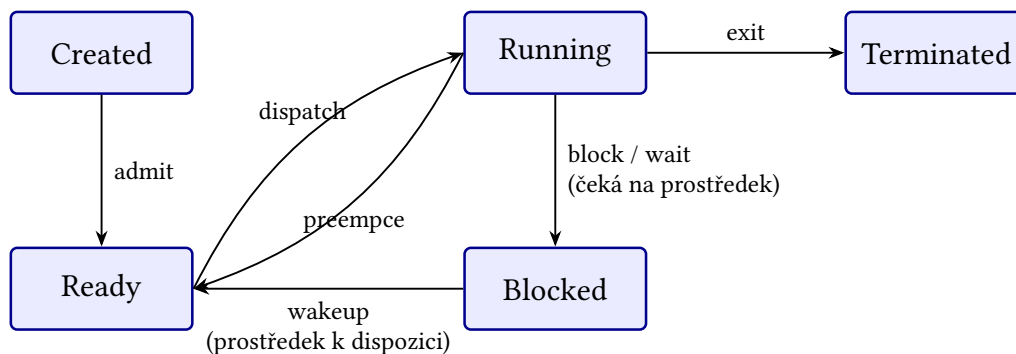
i Poznámka. Linux dnes plánuje implementačně propracovaným algoritmem (dříve CFS, jednotky v červeno-černém stromě indexované nasčítaným časem běhu, plánuje se nejméně využítá). Konkrétní detaily těchto algoritmů jdou nad rámec požadavků okruhu.

5.5 Stavy plánovací jednotky

Vlákno (proces) prochází během života několika *stavy*; přechody mezi nimi vyvolávají události v systému (plánovač, časovač, dostupnost prostředku).

Definice – Stavy a přechody

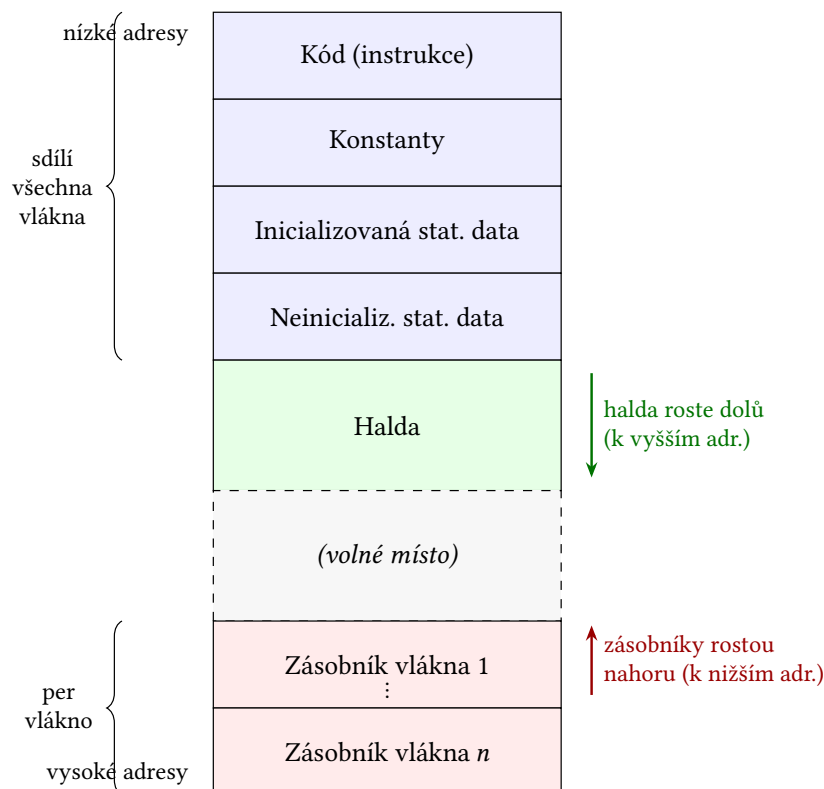
Created (vytvořen) – jednotka vznikla, čeká na přijetí (admission) plánovačem. *Ready* (připraven) – je připravená běžet a čeká *jen na procesor*. *Running* (běžící) – má přidělený procesor a vykonává instrukce. *Blocked* (čeká/spí) – čeká na *prostředek* (dokončení I/O, uvolnění zámku, příchod dat), a tak nemůže pokračovat, ani kdyby procesor dostala. *Terminated* (ukončen) – doběhla; drží se (jako „zombie“), dokud rodič nepřevzme návratový kód. Hlavní přechody: *admit* Created→Ready, *dispatch* Ready→Running, *preemptce* Running→Ready, *block/wait* Running→Blocked, *wakeup* (prostředek k dispozici) Blocked→Ready, *exit* Running→Terminated.



Stavy plánovací jednotky a přechody. Rozdíl *Blocked* vs. *Ready* je klíčový: jednotka v *Blocked* čeká na *prostředek* (typicky dokončení I/O nebo uvolnění zámku) a plánovač jí procesor nepřidělí, dokud prostředek nedorazí (přechod *wakeup*); jednotka v *Ready* je k běhu plně připravená a čeká *jen* na to, až ji plánovač nasadí na procesor. Plánuje se vždy mezi jednotkami ve stavu *Ready*.

5.6 Adresový prostor procesu (kde leží zásobníky vláken)

Než přejdeme k souborům, shrňme rozvržení *adresového prostoru* procesu – ukazuje, co vlákna sdílejí a co má každé vlastní. (Detailní mechanismus mapování na fyzickou paměť je v sekci 6.)



Adresový prostor procesu. Oblasti *kód*, *konstanty*, *inicializovaná* a *neinicializovaná statická data* a *halda* (dynamicky alokovaná paměť, viz sekce 1) jsou pro celý proces *sdílené* mezi vlákny. Naproti tomu *každé vlákno má vlastní zásobník* (lokální proměnné a návratové adresy), protože každé vykonává svou vlastní posloupnost volání. Halda typicky roste směrem k vyšším adresám, zásobníky opačně; mezi nimi je volné místo, do kterého obě strany expandují.

5.7 Soubory: sdílení úložného prostoru

Druhou velkou abstrakcí je sdílení *úložného prostoru*. OS skrývá rozmanitost médií (disk, SSD, síť) za jednotný pojem souboru.

Definice – Soubor

Soubor (file) je pojmenovaná jednotka organizace dat – z pohledu jádra *abstraktní proud* (*posloupnost bajtů*). Jádro *nerozumí* formátu obsahu (to je věc aplikace), vidí jen lineární sled bajtů a jejich délku. Soubor identifikuje OS *číselným* identifikátorem; *jméno a cesta* (ve stromové struktuře adresářů) jsou jen lidsky čitelný odkaz na tento identifikátor.

Intuice. [Analogie s adresovým prostorem] Soubor je vůči datům na disku tím, čím je adresový prostor vůči paměti: *lineární, od nuly adresovatelná posloupnost bajtů*. Místo „adresy“ se v souboru pohybujeme *pozicí* (offsetem) a podobně jako u paměti čteme/zapisujeme souvislé úseky od dané pozice. Operace *seek* (nastavení pozice) je analogie skoku na adresu, sekvenční čtení je analogie procházení paměti po sobě jdoucími adresami. OS tak nabízí jeden jednotný způsob „náhodného přístupu k bajtům“ jak pro paměť, tak pro úložiště.

Definice – Rozhraní pro práci se soubory

Základní operace (POSIX): *open* – otevře soubor podle jména, vrátí *deskriptor*; *read/write* – přečte/zapiše daný počet bajtů od aktuální pozice (a pozici posune); *seek* (*lseek*) – nastaví aktuální pozici v souboru; *close* – soubor zavře a uvolní deskriptor. Doplnkové operace: *create*, *truncate*, *delete*, *flush*, změna atributů.

Definice – File handle / deskriptor

File handle (deskriptor) je číselný identifikátor otevřeného souboru, *lokální pro proces* a přidělovaný postupně. Jádro k němu vede *překládovou tabulku* (deskriptor $\rightarrow$ vnitřní stav otevřeného souboru: o který soubor jde a jaká je aktuální pozice). Aplikace tedy pracuje jen s malým celým číslem, skutečnou vazbu na soubor drží jádro. První tři deskriptory má proces po startu otevřené z konvence: **0 = stdin** (standardní vstup), **1 = stdout** (standardní výstup), **2 = stderr** (chybový výstup).

i Poznámka. Mezi *open* a *read/write* se kvůli výkonu vkládá *vyrovnávací paměť* (buffering) na více úrovních (jádro, běhové prostředí). Jak je soubor skutečně uložen na médiu (souborové systémy FAT, ext, adresáře, bloky) je už věc implementace souborového systému a jde nad rámec tohoto okruhu – nás zajímá souborová *abstrakce a rozhraní*.

5.8 Řešený příklad: plánování Round Robin

Příklad – Round Robin – Ganttův diagram, obrátková a čekací doba

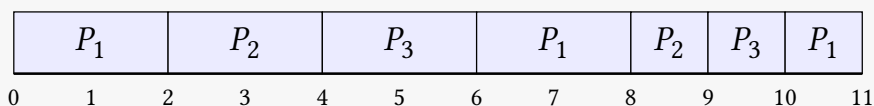
Zadání.

Tři procesy přijdou všechny v čase 0 s dobami běhu $P_1 = 5$, $P_2 = 3$, $P_3 = 3$ (v jednotkách času). Plánuje se algoritmem *Round Robin* s kvantem $q = 2$; ve frontě připravených jsou na začátku v pořadí P_1, P_2, P_3 . Sestrojte Ganttův diagram pořadí běhu a spočítejte pro každý proces *obrátkovou* dobu (turnaround = dokončení – příchod) a *čekací* dobu (waiting = obrátková – doba běhu).

Řešení.

Každý proces běží nejvýš $q = 2$ jednotky, pak (pokud nedoběhl) jde na konec fronty. Procesy přišly v čase 0, takže se po preempci vracejí na konec aktuální fronty. Zbývající doby běhu a průběh fronty:

- [0, 2]: běží P_1 (zbývá $5 \rightarrow 3$), na konec. Fronta: P_2, P_3, P_1 .
- [2, 4]: běží P_2 (zbývá $3 \rightarrow 1$), na konec. Fronta: P_3, P_1, P_2 .
- [4, 6]: běží P_3 (zbývá $3 \rightarrow 1$), na konec. Fronta: P_1, P_2, P_3 .
- [6, 8]: běží P_1 (zbývá $3 \rightarrow 1$), na konec. Fronta: P_2, P_3, P_1 .
- [8, 9]: běží P_2 (zbývá $1 \rightarrow 0$), **dokončen v čase 9**. Fronta: P_3, P_1 .
- [9, 10]: běží P_3 (zbývá $1 \rightarrow 0$), **dokončen v čase 10**. Fronta: P_1 .
- [10, 11]: běží P_1 (zbývá $1 \rightarrow 0$), **dokončen v čase 11**.



Ganttův diagram běhu Round Robin ($q = 2$). Procesy se o procesor střídají po kvantech; P_2 a P_3 doběhnou v poslední polovině (kratší zbytky 1), P_1 s nejdelší dobou běhu doběhne až úplně poslední v čase 11.

Dokončení: P_1 v 11, P_2 v 9, P_3 v 10. Příchod všech = 0, takže obrátková doba = čas dokončení a čekací = obrátková – doba běhu:

proces	doba běhu	dokončení	obrátková	čekací
P_1	5	11	11	$11 - 5 = 6$
P_2	3	9	9	$9 - 3 = 6$
P_3	3	10	10	$10 - 3 = 7$

Průměrná obrátková doba je $\frac{11+9+10}{3} = 10$, průměrná čekací doba $\frac{6+6+7}{3} = \frac{19}{3} \approx 6,33$. Kontrola: součet všech dob běhu je $5 + 3 + 3 = 11$, což přesně sedí s okamžikem, kdy doběhl poslední proces (procesor nikdy nezahálel). (Ilustrativní příklad; Ganttův diagram i doby ověřeny simulací RR v Pythonu.)

6 Virtuální paměť a překlad adres

Procesy nepracují s fyzickými adresami operační paměti přímo, ale se svým vlastním *virtuálním* adresovým prostorem. Mezi ně vstupuje překlad adres, který je jádrem ochrany a sdílení paměti. Požadavky chtějí umět rozlišit virtuální a fyzickou adresu, vysvětlit komponenty adresy (číslo stránky, číslo rámce, offset), *spočítat* překlad nad jednoúrovňovou stránkovací tabulkou a vysvětlit roli virtuálních prostorů v ochraně paměti.

6.1 Virtuální vs. fyzická adresa

Definice – Virtuální a fyzická adresa

Fyzická adresa (FA) adresuje skutečnou buňku operační paměti; používá ji paměťový řadič. *Virtuální adresa* (VA) je adresa ve virtuálním adresovém prostoru procesu; používají ji *všechny* přístupy z instrukcí (včetně načítání samotných instrukcí). Překlad VA na FA provádí hardwarová jednotka MMU (Memory Management Unit) v procesoru podle datových struktur, které pro každý proces připraví operační systém. Pokud pro danou VA mapování neexistuje, vznikne výjimka (*výpadek stránky*, page fault).

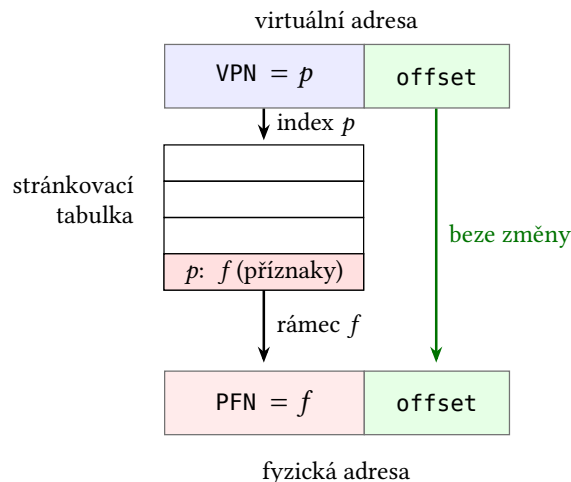
 **Intuice.** Proč vůbec překládat? (1) **Ochrana:** každý proces má vlastní mapování, takže se navzájem „nevidí“ a nemohou si přepsat paměť. (2) **Jednoduchý model:** proces se tváří, že má souvislou paměť jen pro sebe, ač je fyzicky rozházená po rámcích. (3) **Iluze velikosti:** virtuální prostor může být větší než fyzická paměť (nepoužité stránky odložené na disk). Otázka „je tahle adresa virtuální, nebo fyzická?“ se pozná podle toho, kdo ji vidí: cokoli v kódu procesu (ukazatele, cíle skoků, operandy load/store) je *virtuální*; teprve výstup MMU, který jde na sběrnici do paměti, je *fyzický*. Výjimkou jsou pole stránkovací tabulky – ta obsahují *fyzická* čísla rámců, protože sama MMU je čte z fyzické paměti. *Příklad rozhodování:* v ovladači disku (sekce 4) je parametr `register_address` *virtuální* (ukazatel v kódu, přes který přistupujeme k mapovaným registrům), kdežto `data_phys_addr` předávaná DMA řadiči je *fyzická* (řadič ji klade přímo na sběrnici, mimo MMU).

6.2 Stránkování: stránky, rámce, stránkovací tabulka

Nejběžnějším mechanismem překladu je *stránkování* (paging).

Definice – Stránka, rámec, stránkovací tabulka

Virtuální prostor je rozdělen na *stránky* (pages) o pevné velikosti 2^k bajtů, fyzický prostor na stejně velké *rámce* (frames). Jedna stránka se vejde přesně do jednoho rámce. Virtuální adresa se rozpadá na *číslo stránky* (VPN, horní bity) a *offset* (dolních k bitů, pozice uvnitř stránky). *Stránkovací tabulka* (page table) je pole indexované číslem stránky; každá položka (PTE) obsahuje *číslo rámce* (PFN) a příznaky (alespoň bit platnosti). Tabulka leží v paměti zvlášť pro každý proces.



Překlad virtuální adresy jednoúrovňovou stránkovací tabulkou. Horní bity VA (číslo stránky p) indexují tabulku; přečtená položka dá číslo rámce f . Fyzická adresa vznikne *spojením* čísla rámce a původního offsetu – offset se nepřekládá (poloha uvnitř stránky i rámce je stejná). Není-li položka platná, nastane výpadek stránky.

6.3 Překlad adresy nad jednoúrovňovou tabulkou

Postup překladu VA na FA má tři kroky, které stačí umět spočítat ručně:

1. **Rozlož VA** na číslo stránky a offset. Při velikosti stránky 2^k je offset dolních k bitů ($\text{offset} = \text{VA} \bmod 2^k$), číslo stránky zbytek ($\text{VPN} = \lfloor \text{VA} / 2^k \rfloor$).
2. **Najdi položku** ve stránkovací tabulce: její adresa je báze tabulky + $\text{VPN} \cdot (\text{velikost položky})$. Přečti položku; pokud je neplatná, nastane výpadek stránky.
3. **Sestav FA** spojením čísla rámce z položky a původního offsetu: $\text{FA} = (\text{PFN} \cdot 2^k) + \text{offset}$.

i Poznámka. Jednoúrovňová tabulka pro 32-bit adresy a 4 KiB stránky má 2^{20} položek po 4 B, tj. 4 MiB na proces – proto se v praxi používají víceúrovňové tabulky (chybějící podtabulky se neukládají) a překlad se urychluje TLB (níže). Požadavky okruhu i řešené příklady pracují s jednoúrovňovou tabulkou.

Příklad – SZZ léto 2023 č. 17: Překlad adresy stránkováním

(Reálná otázka SZZ: léto 2023, č. 17 – Překlad adresy (stránkování))

(Pozn.: v rámci současné akreditace nešlo o společnou otázku ($q1-q4$), ale o otázku jiné specializace (specializační sekce, systémové zaměření, ne UI). Do společného okruhu I4 tedy formálně nepatří; tematicky sem ovšem spadá a je výborná k procvičení.)

Zadání.

Hypotetický procesor, 16-bitový virtuální i fyzický prostor, 1-úrovňové stránkování s 64 B stránkami. Záznamy tabulky jsou 16-bitové; offsetové bity slouží jako příznaky, *nejméně významný bit* je příznak platnosti (1 = platný). Stránkovací tabulka je na adrese 0x1d40 (její výpis je ve složce příklady/ jako oříznutá tabulka z PDF). Přelož virtuální adresu 0x328f (= 0b0011001010001111): (1) číslo stránky a offset, (2) fyzickou adresu položky tabulky, (3) výslednou fyzickou adresu.

Řešení.

(1) **Rozklad VA.** Stránka má $64 = 2^6$ B, takže offset jsou dolní 6 bitů, číslo stránky horních 10 bitů:

$$0x328f = \underbrace{0b0011001010}_{\text{číslo stránky}} \underbrace{001111}_{\text{offset}}.$$

Číslo stránky = 0b0011001010 = 0xCA = 202, offset = 0b001111 = 0x0F = 15.

(2) **Adresa položky.** Položky jsou 16-bitové, tj. 2 B; položka pro stránku 202 je na adrese

$$0x1d40 + 202 \cdot 2 = 0x1d40 + 0x194 = \boxed{0x1ED4}.$$

(3) **Výsledná FA.** Položka stránkovací tabulky na adrese 0x1ED4 má hodnotu 0xFBF7 (odečteno ze zadaného výpisu tabulky, viz příklady/). Nejnižší bit je 1, položka je tedy *platná*. Číslo rámce tvoří horních 10 bitů (dolních 6 bitů jsou příznaky): $0xFBF7 \& 0xFFC0 = 0xFBC0$. Fyzická adresa vznikne spojením s offsetem (dolních 6 bitů):

$$FA = 0xFBC0 \mid 0x0F = \boxed{0xFBCF}.$$

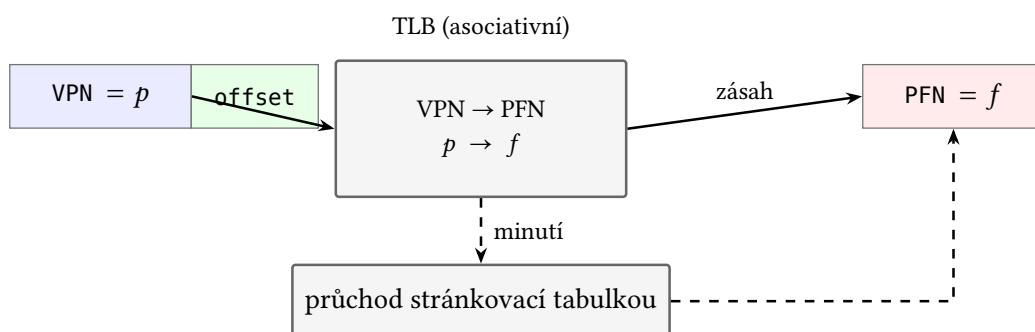
Pozn.: Tato otázka nemá v PDF náčrt řešení; výpočet (vč. odečtu hodnoty 0xFBF7 z tabulky) jsme ověřili skriptem `verify_szz.py`.

6.4 TLB: urychlení překladu

Každý přístup do paměti by jinak vyžadoval *další* přístup do paměti kvůli stránkovací tabulce. Proto má MMU malou rychlou cache překladů.

Definice – TLB (Translation Lookaside Buffer)

TLB je asociativní cache nedávno použitých překladů: klíčem je číslo stránky, hodnotou číslo rámce (a příznaky). Při překladu MMU nejdřív zkusí TLB; při *zásahu* (hit) získá rámec rovnou, při *minutí* (miss) projde stránkovací tabulku a výsledek do TLB uloží. TLB využívá časovou a prostorovou lokalitu (kód i data se chvíli drží v téže stránce).



Překlad s TLB. Při zásahu dá TLB číslo rámce f okamžitě (bez přístupu do paměti). Při minutí se projde stránkovací tabulka a nalezený překlad se do TLB doplní. Offset se opět nepřekládá. Při více procesech musí TLB rozlišovat adresové prostory – buď se při přepnutí kontextu vyprázdní, nebo každá položka nese identifikátor prostoru (ASID).

Příklad – SZZ léto 2023 č. 15: Stránkování a TLB

(Reálná otázka SZZ: léto 2023, č. 15 – Stránkování (TLB))

(Pozn.: v rámci současné akreditace nešlo o společnou otázku ($q1-q4$), ale o otázku jiné specializace (specializační sekce, systémové zaměření, ne UI). Do společného okruhu I4 tedy formálně nepatří; tematicky sem ovšem spadá a je výborná k procvičení.)

Zadání.

Procesor s 32-bitovými (4 B) virtuálními i fyzickými adresami, softwarově plněný TLB, pevná velikost stránky 4 MB. Položka TLB má 4 B a obsahuje (od nejvyšších bitů) potřebnou část virtuální a fyzické adresy, ASID (identifikátor adresového prostoru), a bity *dirty* a *valid*; dva nejnižší bity jsou rezervované (0). Aktuální ASID je 0x12. Obsah TLB (devět 32-bitových položek):

0x35C0C124	0x35C0613C	0x35C08120
0x36005134	0x36009128	0x3600D12C

0x3640E124 0x3640A120 0x3640413C

Co se stane při přístupech 0x35803580 (zápis), 0x35CD35CD (čtení), 0x36003600 (čtení), 0x36403640 (zápis)?

Řešení.

Formát položky. Stránka má 4 MB = 2^{22} B, takže offset je 22 bitů a číslo stránky (i rámce) má $32 - 22 = 10$ bitů. Položka má 32 bitů: od nejvyšších 10 bitů VPN, 10 bitů PFN, pak ASID, dirty, valid a 2 rezervované; na ASID tedy zbývá $32 - 10 - 10 - 1 - 1 - 2 = 8$ bitů.

31 ... 22	21 ... 12	11 ... 4	3	2	1 0
VPN (10)	PFN (10)	ASID (8)	D	V	00

Formát 32-bitové položky TLB: VPN [31 : 22], PFN [21 : 12], ASID [11 : 4], dirty [3], valid [2], rezervované [1 : 0]. Dekódování položky: např. 0x35C0C124 dá VPN 0x0D7, PFN 0x00C, ASID 0x12, D=0, V=1.

Vyhodnocení přístupů. U každé adresy vezmeme horních 10 bitů jako VPN a hledáme v TLB položku se *shodným VPN*, *platnou* (V=1) a se *shodným ASID* (0x12); jinak je to minutí. Fyzická adresa je $\text{PFN} \cdot 2^{22} + \text{offset}$, kde offset jsou dolní 22 bity VA.

1. write 0x35803580: VPN = 0x0D6. Žádná položka s tímto VPN v TLB *není* $\Rightarrow$ **minutí TLB**. Protože je TLB plněn softwarově, procesor vyvolá výjimku „TLB miss“; obslužná rutina OS dohledá překlad ve stránkovací tabulce a doplní TLB (a instrukci zopakuje), případně – pokud mapování neexistuje – vyvolá výpadek stránky. Výslednou FA nelze z TLB určit.
2. read 0x35CD35CD: VPN = 0x0D7. Tři položky mají toto VPN, ale jen 0x35C0C124 má V=1 a ASID 0x12 (ostatní mají ASID 0x13, resp. V=0). PFN = 0x00C, offset = 0x0D35CD, tedy FA = 0x030D35CD.
3. read 0x36003600: VPN = 0x0D8. Platná položka se správným ASID je 0x3600D12C (PFN 0x00D); položka 0x36009128 má sice ASID 0x12, ale V=0, a 0x36005134 má ASID 0x13. Offset = 0x003600, tedy FA = 0x03403600.
4. write 0x36403640: VPN = 0x0D9. Platná položka se správným ASID je 0x3640E124 (PFN 0x00E, D=0). Offset = 0x003640, překlad tedy vede na FA = 0x03803640. Jde ale o *zápis* na stránku s D=0, takže zápis hned neproběhne: procesor vyvolá výjimku (u MIPS „TLB modified“). OS ověří, že stránka je zapisovatelná, nastaví v položce D=1 (a stránku si poznamená jako změněnou) a instrukci zopakuje; zápis pak proběhne na uvedenou FA. Není-li stránka zapisovatelná, jde o porušení ochrany.

Pozn.: Tato otázka nemá v PDF náčrt řešení; dekódování položek i všechny čtyři překlady jsme ověřili skriptem verify_szz.py. Poučení: položky s jiným ASID nebo V=0 se nesmí použít – právě ASID umožňuje držet v TLB překlady více procesů zároveň.

6.5 Ochrana paměti

Definice – Ochrana paměti virtuálními adresovými prostory

Každý proces má *vlastní* stránkovací tabulku, tedy vlastní mapování VA na rámce. Stránka jednoho procesu se mapuje na jiné rámce než stejně číslovaná stránka jiného procesu, takže se procesy svými virtuálními adresami *nemohou* dostat do cizí paměti – přístup mimo namapovanou oblast skončí výpadkem stránky. Příznaky v položce navíc omezují způsob přístupu (jen pro čtení, neexecuovatelné). Vlákná téhož procesu naopak *sdílejí* jeden adresový prostor (jednu tabulku), proto se musí synchronizovat (sekce 7).

Intuice. Ochrana je „zadarmo“ jako vedlejší efekt překladu: stačí, že OS do tabulky procesu nevloží žádné mapování na cizí rámce. Sdílení paměti mezi procesy se pak dělá naopak záměrně – OS namapuje tytéž rámce do tabulek dvou procesů (typicky na různé virtuální adresy). Přepnutí kontextu mezi procesy proto zahrnuje i výměnu stránkovací tabulky (a většinou vyprázdnění nebo přeznačení TLB).

Příklad – SZZ podzim 2022 č. 2: Virtuální paměť – stránkování 32-bit

(Reálná otázka SZZ: podzim 2022, č. 2 – Virtuální paměť (stránkování, 32-bit))

Zadání.

Procesor se stránkováním, 32-bit virtuální i fyzická adresa, jednoúrovňová stránkovací tabulka, stránka 4 KiB. **(A)** Navrhněte formát položky tabulky (PTE) a spočítejte, kolik položek tabulka má a kolik paměti zabere. **(B)** V pseudokódu napište `create_pte(phys_addr, present, writable, executable)` a `set_mapping(page_table, virt_addr, entry)`. **(C)** Souvislý blok 4 stránek od `0x0000B000` (heap: čte se a modifikuje, nelze spouštět) je namapován na souvislý blok rámců; čtení 4 B z `0x0000CE90` se přeložilo na `0xA1018E90`, zápis proběhl na `0x0000D854`. Tabulka leží od `0x13400000`. Určete fyzické adresy položek pro tento blok a jejich hexadecimální hodnoty včetně atributů.

Řešení.

(A) Rozměry a PTE. Stránka má $4 \text{ KiB} = 2^{12} \text{ B}$, takže offset jsou dolní 12 bitů a číslo stránky horních $32 - 12 = 20$ bitů. Tabulka má proto $2^{20} = 1\,048\,576$ položek; při 4 B na položku zabere $2^{20} \cdot 4 \text{ B} = 4 \text{ MiB}$ na proces. Číslo rámce (PFN) má rovněž 20 bitů a vejde se do horních bitů [31 : 12]; dolních 12 bitů zbývá na příznaky. Zvolíme rozložení: bity [31 : 12] PFN, bit 6 **D** (Dirty), bit 5 **A** (Accessed), bit 2 **X** (eXecutable), bit 1 **W** (Writable), bit 0 **P** (Present); A a D nastavuje hardware, ostatní OS. 4-bytová položka stačí na jedno čtení z paměti.

(B) Pseudokód.

```
function create_pte(phys_addr, present, writable, executable):
    pte = phys_addr & 0xFFFFF000           // PFN v horních 20 bitech
    if present:    pte = pte | 0x1         // bit 0: Present
    if writable:   pte = pte | 0x2         // bit 1: Writable
    if executable: pte = pte | 0x4         // bit 2: eXecutable
    return pte

procedure set_mapping(page_table, virt_addr, entry):
    index = virt_addr >> 12                // číslo stránky = horních 20 bitů
    page_table[index] = entry
```

(C) Překlad a položky. Čtená VA `0x0000CE90`: stránka `0xC`, offset `0xE90`; FA `0xA1018E90` má stejný offset a rámec `0xA1018`. Blok 4 stránek od `0x0000B000` jsou stránky `0xB`, `0xC`, `0xD`, `0xE`; mapování je souvislé, takže rámce navazují:

$0xB \rightarrow 0xA1017, \quad 0xC \rightarrow 0xA1018, \quad 0xD \rightarrow 0xA1019, \quad 0xE \rightarrow 0xA101A.$

Adresy položek (báze `0x13400000` + číslo stránky $\cdot 4$):

$0xB \rightarrow 0x1340002C, \quad 0xC \rightarrow 0x13400030, \quad 0xD \rightarrow 0x13400034, \quad 0xE \rightarrow 0x13400038.$

Příznaky: heap $\Rightarrow$ u všech $P=1, W=1, X=0$ (příznakový byte `0x03`). Čtení ze stránky `0xC` $\Rightarrow A=1$ (přidá bit 5, `0x23`); zápis na `0x0000D854` (stránka `0xD`) $\Rightarrow A=1$ i $D=1$ (přidá bit 6, `0x63`); stránky `0xB`, `0xE` nepoužity. Hodnota položky = $(\text{PFN} \ll 12) | \text{příznaky}$:

$0xB \rightarrow 0xA1017003, \quad 0xC \rightarrow 0xA1018023, \quad 0xD \rightarrow 0xA1019063, \quad 0xE \rightarrow 0xA101A003.$

Pozn.: Tato otázka nemá v PDF náčrt řešení; číselné výsledky jsme ověřili numericky (viz příklady/). Příznakové byty závisejí na zvoleném rozložení PTE; PFN a adresy položek na něm nezávisejí.

7 Paralelismus a synchronizace

Vlákna téhož procesu sdílejí adresový prostor (sekce 6), takže mohou současně sahát na stejná data. Bez koordinace vznikají *časově závislé chyby*. Požadavky chtějí umět rozpoznat race condition, vymezit kritickou sekci a vzájemné vyloučení a použít základní synchronizační primitiva (zámky, semaforey) včetně rozdílu aktivního a pasivního čekání. Řešené příklady na konci sekce procvičují obojí: *nalezení chyby* i *správnou konstrukci* synchronizace.

7.1 Paralelismus a souběžnost

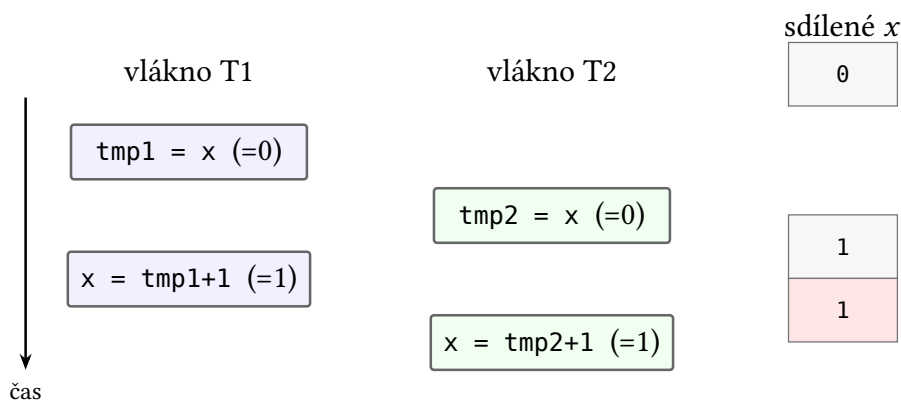
Definice – Paralelní vs. souběžný výpočet

Paralelní (parallel) výpočet probíhá fyzicky současně na více jádrech. *Souběžný* (concurrent) výpočet znamená, že více činností je „rozpracováno“ zároveň, ale nemusí běžet fyzicky současně (na jednom jádru se prokládají multitaskingem). Problémy synchronizace vznikají v obou případech, kdykoli více vláken *sdílí* data.

7.2 Časově závislé chyby (race conditions)

Definice – Race condition

Časově závislá chyba (race condition) nastane, když výsledek výpočtu závisí na *pořadí* nebo *časování* prokládání vláken, která přistupují (a zapisují) ke sdíleným datům. Typickým zdrojem je operace „přečti – uprav – zapiš“ (např. $x = x + 1$), která není atomická: mezi čtení a zápis se může vklínit jiné vlákno.



Ztracená aktualizace (lost update) při neatomickém $x = x + 1$. Obě vlákna přečtou $x = 0$ dříve, než kterékoli zapiše, takže obě zapišou 1; jeden přírůstek se ztratí (správně mělo být $x = 2$). Výsledek závisí na prokládání – proto „časově závislá chyba“.

7.3 Kritická sekce a vzájemné vyloučení

Definice – Kritická sekce, vzájemné vyloučení

Kritická sekce je úsek programu, kde se přistupuje ke sdílenému prostředku a kde by souběžný přístup způsobil chybu. *Vzájemné vyloučení* (mutual exclusion) je požadavek, aby kritickou sekci nad tímž prostředkem vykonávalo v jednu chvíli *nejvýše jedno* vlákno. Synchronizace kritickou sekci „ohradí“ tak, aby vlákna do ní vstupovala po jednom.

Intuice. Co musí splňovat správné řešení. Samotné vzájemné vyloučení nestačí; od korektního ošetření kritické sekce chceme tři vlastnosti:

- (1) *Vzájemné vyloučení* (mutual exclusion) – v kritické sekci je nejvýše jedno vlákno (*bezpečnost*: nikdy se nestane nic špatného).
- (2) *Trvalý postup* (progress) – není-li v sekci nikdo a některá vlákna chtějí vstoupit, musí být v konečném čase některé vpuštěno; o vpuštění nesmí rozhodovat vlákno, které samo o vstup neusiluje. Porušením je např. *uváznutí* (deadlock).
- (3) *Omezené čekání* (bounded waiting) – existuje mez, kolikrát smí jiná vlákna vstoupit do sekce dříve než vlákno, které už o vstup požádalo. Bez ní hrozí *vyhladovění* (starvation): vlákno čeká neomezeně dlouho, i když se sekce stále uvolňuje.

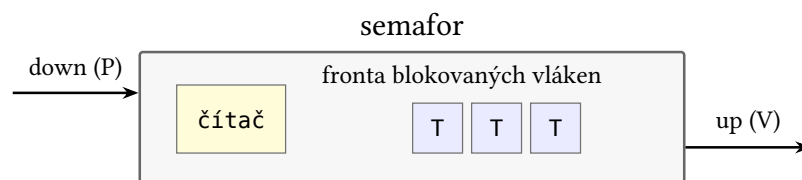
7.4 Synchronizační primitiva

Definice – Aktivní vs. pasivní čekání

Aktivní čekání (busy waiting): vlákno čeká v cyklu a opakovaně testuje podmínku, čímž *spotřebovává* procesor (např. spin-lock). Vhodné jen pro velmi krátká čekání. *Pasivní (blokující) čekání*: vlákno je zablokováno (vyřazeno z plánování) a OS ho probudí, až bude moci pokračovat – procesor zatím dělá něco jiného. Vhodné pro delší čekání.

Definice – Zámek, semafor, mutex

Zámek chrání kritickou sekci: operace *lock* ji uzamkne (čeká, je-li již zamčeno), *unlock* odemkne. *Spin-lock* implementuje lock aktivním čekáním nad atomickou instrukcí (test-and-set / compare-and-swap). *Semafor* je chráněný čítač s frontou čekajících vláken a atomickými operacemi *down* (P: je-li čítač > 0, sniž ho, jinak vlákno zablokuj a zařaď do fronty) a *up* (V: je-li někdo ve frontě, probud ho, jinak zvyš čítač). *Mutex* je semafor s čítačem 1 (binární), tj. vzájemné vyloučení pro jedno vlákno.



Semafor = čítač + fronta blokovanych vláken (T). *down* buď čítač sniží (je-li kladný), nebo vlákno *pasivně* zablokuje. *up* buď probudí čekající vlákno z fronty, nebo (je-li prázdná) zvýší čítač. Mutex je binární semafor (čítač 1). Obě operace musí být atomické.

Poznámka. Uváznutí (deadlock). Skupina vláken může uváznout, čeká-li každé na prostředek držený jiným z nich. Coffmanovy nutné podmínky: vzájemné vyloučení, „drž a čekej“, neodnímatelnost prostředků a cyklické čekání. Klasický příklad: dvě vlákna zamykají dva mutexy v opačném pořadí. Tomu se vyhneme např. zamykáním vždy ve stejném pořadí. Deadlock není v doslovných požadavcích jmenován, ale objevuje se v řešených státnicových otázkách (semafory), proto ho zde zmiňujeme.

7.5 Řešené příklady

Příklad – SZZ podzim 2023 č. 1: Race condition a kritická sekce

(Reálná otázka SZZ: podzim 2023, č. 1 – Architektura PC a OS (race condition, kritická sekce))

Zadání.

Třída `LimitedStack` s metodami `getDepth` (řádek 10: `return currentDepth;`) a `pushValue` (řádky 14–17):

```
bool pushValue(int v) {           // 13
    if (currentDepth >= maxDepth) // 14
        return false;
    stack[currentDepth] = v;      // 16
    currentDepth++;              // 17
    return true;
}
```

Více vláken volá `getDepth`/`pushValue` nad sdílenou instancí. (1) Může nastat race condition? Které řádky tvoří kritickou sekci? (2) Může přetéct zásobník (zápis mimo pole)? (3) Odstraňte problém.

Řešení.

- (1) **Ano.** Kritickou sekci jsou řádky 10 a 14–17: čtení i aktualizace sdílené `currentDepth` (a zápis do `stack`). Operace `currentDepth++` je „přečti–uprav–zapiš“, navíc test na řádku 14 a zápis na 16–17 nejsou vůči jiným vláknům atomické.
- (2) **Ano, může.** Dvě vlákna `T1`, `T2` vykonají *souběžně* test na řádku 14 při `currentDepth = maxDepth - 1`; obě projdou (test je nepřekročen). `T2` je pak pozastaveno, `T1` dokončí zápis a zvýší `currentDepth` na `maxDepth`. Když `T2` pokračuje, zapiše na index `currentDepth`, který už ukazuje za pole – přetečení.
- (3) **Vzájemné vyloučení** nad celou kritickou sekci. V Javě stačí `synchronized` metody; v C# `lock` nad pomocným objektem; v C++ `std::mutex` a `std::lock_guard` (RAII zajistí odemčení i při `return`). Důležité je zamknout už *před* testem na řádku 14 a odemknout až po inkrementaci.

```
bool pushValue(int v) {
    std::lock_guard<std::mutex> g(mtx); // lock; unlock at scope exit
    if (currentDepth >= maxDepth) return false;
    stack[currentDepth] = v;
    currentDepth++;
    return true;
}

int getDepth() {
    std::lock_guard<std::mutex> g(mtx); // ctení sdíleného stavu rovněž zamknout
    return currentDepth;
}
```

Příklad — SZZ léto 2023 č. 14: Semafor a producent–konzument

(Reálná otázka SZZ: léto 2023, č. 14 – Synchronizace)

(Pozn.: v rámci současné akreditace nešlo o společnou otázku (q1–q4), ale o otázku jiné specializace (specializační sekce, systémové zaměření, ne UI). Do společného okruhu I4 tedy formálně nepatří; tematicky sem ovšem spadá a je výborná k procvičení.)

Zadání.

(1) Načrtněte rozhraní semaforu a sémantiku operací. (2) Pomocí něj implementujte producent–konzument s vyrovnávací pamětí (warehouse) o kapacitě `capacity`. Datová struktura pro uložení vyprodukovaných položek má jen rozhraní (`push/pop`) a *sama* žádnou synchronizaci negarantuje.

Řešení.

(1) **Rozhraní semaforu.** Semafor drží nezáporný čítač a frontu čekajících vláken; `down()` (P) čítač snižuje, nebo vlákno zablokuje (je-li čítač 0); `up()` (V) probudí čekající vlákno, nebo čítač zvýší. Obě operace jsou atomické (viz výklad výše).

(2) **Producent–konzument.** Použijeme *tři* semafore: `free` (počet volných míst, `init capacity`), `items` (počet připravených položek, `init 0`) a binární `mtx` (vzájemné vyloučení nad bufferem, `init 1`). Producent počká na volné místo, konzument na položku; přístup k bufferu chrání `mtx`.

```
typedef struct {
    semaphore free; // init = capacity (volna mista)
    semaphore items; // init = 0 (pripravene polozky)
    semaphore mtx; // init = 1 (vzajemne vyloucení nad bufferem)
```

```

    buffer_t buf;          // push/pop bez vlastní synchronizace
} prodcons_t;

void prodcons_produce(prodcons_t *pc, void *item) {
    down(&pc->free);        // počkej na volné místo
    down(&pc->mtx);          // vstup do kritické sekce
    buffer_push(&pc->buf, item);
    up(&pc->mtx);            // opust kritickou sekci
    up(&pc->items);          // ohlas novou položku
}

void *prodcons_consume(prodcons_t *pc) {
    down(&pc->items);        // počkej na položku
    down(&pc->mtx);
    void *item = buffer_pop(&pc->buf);
    up(&pc->mtx);
    up(&pc->free);          // uvolni místo
    return item;
}

```

Proč to funguje. Invariant $free = capacity - \text{počet položek}$, $items = \text{počet položek}$: producent nikdy nezapíše do plného bufferu (čeká na $free$), konzument nikdy nečte z prázdného (čeká na $items$). mtx brání souběžnému push/pop. Pořadí je důležité: počítací semafor se získává *před* mtx , jinak by hrozil deadlock (vlákno by drželo mtx a čekalo na místo, které nikdo neuvolní). mtx je potřeba i při jediném producentovi a jediném konzumentovi: push a pop mohou běžet současně nad touž nesynchronizovanou strukturou. Bez zámku by to šlo jen s bufferem navrženým právě pro jednoho producenta a jednoho konzumenta (kruhový buffer, kde každá strana mění jen svůj index), což zadání nezaručuje. (V PDF bez náčrtu řešení; správnost konstrukce – dodržení invariantů a nepřítomnost deadlocku – jsme ověřili rozбором pořadí operací.)

Příklad — SZZ léto 2024 č. 2: Semafor, race condition, deadlock

(Reálná otázka SZZ: léto 2024, č. 2 – Semafor (P/V, race condition, deadlock))

Zadání.

Dvě vlákna mají vypsát OneTwoThreeFourFive (liché části f1, sudé f2) přes *jeden* binární semafor `sem` (init 0, WaitOne=P, Release=V). Synchronizace „skoro funguje“, ale má race condition. (1) Najděte scénář s jiným výstupem nebo deadlockem (zapsaný jako posloupnost dokončených řádků). (2) Pomohlo by FIFO pořadí probouzení? (3) Opravte to dvěma semafori.

```

void f1(){ Write("One");   /*7*/ sem.Release(); /*8*/ sem.WaitOne(); /*9*/
           Write("Three"); /*10*/ sem.Release(); /*11*/ sem.WaitOne(); /*12*/
           Write("Five");  /*13*/ }
void f2(){ sem.WaitOne(); /*18*/ Write("Two");   /*19*/ sem.Release(); /*20*/
           sem.WaitOne(); /*21*/ Write("Four");  /*22*/ sem.Release(); /*23*/ }

```

Řešení.

- (1) **Chybný scénář.** Problém: f1 po Release (řádek 8) hned WaitOne (řádek 9) a *sebere si vlastní* token. Posloupnost 7-8-9-10-11-12-13 dá výstup `OneThreeFive`; f2 pak navždy uvázne na řádku 18 ($sem = 0$), tj. *deadlock*. (Stačí uvést jeden takový scénář.)
- (2) **Ne, nepomohlo.** Chyba není v pořadí probouzení, ale v tom, že f1 si může token vzít sama dřív, než se f2 vůbec dostane k WaitOne. Scénář (1) funguje i při FIFO.
- (3) **Oprava dvěma semafori** `semA`, `semB` (oba init 0): každé vlákno *signalizuje druhému* (do jiného semaforu, než z jakého čeká), takže si nemůže vzít vlastní token.

```

void f1(){ Write("One");   semA.Release(); semB.WaitOne();
           Write("Three"); semA.Release(); semB.WaitOne();
           Write("Five");  }
void f2(){ semA.WaitOne(); Write("Two");   semB.Release();
           semA.WaitOne(); Write("Four");  semB.Release(); }

```

Výstup je vždy `OneTwoThreeFourFive`: f2 čeká na semA (plní f1), f1 čeká na semB (plní f2), výpisy se tak střídají v pevném pořadí.

Příklad — SZZ jaro 2026 č. 2: Semafor mezi procesy (FIFO buffer)

(Reálná otázka SZZ: jaro 2026, č. 2 – Semafor mezi procesy (FIFO buffer))

Zadání.

Jednosměrný proud bajtů mezi dvěma procesy přes FIFO buffer kapacity N . `send(ch)` počká na volné místo a vloží bajt, `receive()` počká na bajt a odebere ho. K dispozici jsou jen *semafony přes hranice procesů* a atomické `buffer_push`/`buffer_pop` (bez synchronizace, nesmí se volat na plný/prázdný buffer). *Bez aktivního čekání.* (1) Implementujte `send`/`receive` a uveďte počáteční stavy semaforů. (2) Uveďte invarianty semaforů vůči počtu prvků. (3) Vysvětlete, jak se invarianty mění uvnitř funkcí.

Řešení.

(1) Implementace. Dva semafony: `capacity` (volná místa, `init N`) a `content` (platné bajty, `init 0`). Protože jsou `buffer_push`/`buffer_pop` atomické a funkce běží v různých procesech, není potřeba další mutex.

```
// capacity init = N, content init = 0
void send(byte ch) {
    capacity.down();    // počkej na volné místo (pasivní čekání)
    buffer_push(ch);
    content.up();       // ohlas nový bajt
}
byte receive() {
    content.down();     // počkej na bajt
    byte ch = buffer_pop();
    capacity.up();     // uvolni místo
    return ch;
}
```

(2) Invarianty (mezi voláními, tj. v klidovém stavu): označíme-li `elems` počet platných bajtů v bufferu, pak

$$\text{capacity} = N - \text{elems}, \quad \text{content} = \text{elems}.$$

Odtud $0 \leq \text{elems} \leq N$, takže `buffer_push` se nikdy nevolá na plný a `buffer_pop` na prázdný buffer.

(3) Změny uvnitř funkcí. V `send` po `capacity.down()` je invariant 1 dočasně „posunut“ o +1 (jedno místo rezervováno, ale ještě nezaplněno); `buffer_push` ho srovná a poruší invariant 2 o –1; `content.up()` srovná i ten. V `receive` symetricky: `content.down()` posune invariant 2, `buffer_pop` ho srovná a poruší invariant 1, `capacity.up()` srovná. Mezi voláními tedy oba invarianty platí. (Tato dekompozice odpovídá oficiálnímu náčrtu řešení.)

Shrnutí

i Poznámka. Klíčové pojmy a vzorce (přehled k SZZ). Stručný výčet klíčových definic, vět a vzorců okruhu I4. Zkratky: VA = virtuální adresa, FA = fyzická adresa, VPN = virtual page number, PFN = page frame number, ISA = instrukční sada.

Architektura a reprezentace dat.

- Von Neumann: program i data v téže paměti; Harvard: oddělené paměti (mikrokontroléry, L1 cache). Bajt = 8 bitů, jednotka adresace; N -bitový adresový prostor = $[0, 2^N)$.
- Little-endian: LSB na nejnižší adrese (x86); big-endian: MSB na nejnižší adrese (TCP/IP). Při čtení hexdumpu zkontroluj, která konvence platí – ovlivní, jak skládat vícebajtová čísla.
- Dvojkový doplněk: rozsah $[-2^{N-1}, 2^{N-1} - 1]$, $-x = \sim x + 1$; jediná nula; sčítání/odčítání = stejný obvod jako unsigned. V 8 b: $0xFF = 255$ (unsigned) nebo -1 (signed); $0x80 = 128$ nebo -128 .

Zápornost: nejvyšší bit = 1.

- IEEE 754 single (32 b): $(-1)^s \cdot (1.f)_2 \cdot 2^{e-127}$, schéma 1+8+23, bias 127; double 1+11+52. Krajní exponent: $e = 0 \rightarrow$ nula/denormalizovaná; $e = 0xFF \rightarrow \pm\infty$ (mantisa 0) nebo NaN.
- Zarovnání: typ délky s začíná na adrese dělitelné s ; struktura zarovnává podle největšího členu $\Rightarrow$ vnitřní + vnější padding; `sizeof` = násobek zarovnání. Pořadí členů od největšího minimalizuje výplň.
- Logický posun ($\ll$, $\gg$ unsigned): doplní nuly; aritmetický posun vpravo (signed): doplní znaménkový bit. Posun vlevo o k = násobení 2^k , vpravo = celočíselné dělení 2^k (aritmetický posun zaokrouhluje dolů, tj. u záporných jinak než / v C).

Instrukční sada a vazba na vyšší jazyky.

- ISA = kontrakt překladač/hardware; assembler = symbolický zápis strojového kódu. MIPS: pevná délka 4 B, R-type 6+5+5+5+5+6 b (op, rs, rt, rd, shamt, funct). x86: proměnná délka 1–15 B.
- RISC (MIPS/ARM/RISC-V): málo instrukcí, pevná délka, aritmetika jen mezi registry, přístup do paměti jen load/store; snadné zřetězení. CISC (x86): mnoho instrukcí proměnné délky, operace i nad pamětí; kompaktnější kód.
- Přiřazení $\rightarrow$ load-imm + store. Podmínka $\rightarrow$ obrácený test + podmíněný skok do else + nepodmíněný skok za celou podmínku. Cyklus $\rightarrow$ inicializace + skok na test + tělo + aktualizace + zpětný skok. Rozpoznání zpětného skoku = cyklus.
- Volání: předej parametry (registry a0–a3/zásobník), jal f uloží NA do ra a skočí, jr ra se vrátí. ABI určuje, kde jsou argumenty, kde návratová hodnota (v0), kdo schovává registry.
- Zásobníkový rámeček (roste k nižším adresám): argumenty | návratová adresa | starý FP | lokální proměnné; FP = začátek rámce, SP = vrchol. Offset do pole: $i \cdot \text{sizeof}(\text{prvku})$; `slld r, i, 2` = násobení 4 pro int.

Podpora OS (privilegované režimy, jádro).

- Uživatelský režim (ring 3): bez přístupu k HW, bez privilegovaných instrukcí; pokus o ně $\rightarrow$ výjimka. Jádrový (ring 0): plný přístup. Na x86: 4 ringy, v praxi jen 0 a 3.
- Syscall: instrukce `syscall` (nebo `int 0x80`) $\rightarrow$ přepne do ring 0 + skok na pevný vstupní bod (adresa v `IA32_LSTAR`); obsluha; `sysret` $\rightarrow$ ring 3. Cíl skoku určuje *jádro*, ne aplikace – to je jádro ochrany.
- Architektury jádra: monolit (subsystémy jako funkce, stejná oprávnění; Linux), vrstvené (hierarchie vrstev), mikrojádro (minimum v ring 0, subsystémy jako user-mode servery komunikující zprávami; MINIX, QNX – větší bezpečnost, vyšší režie).

Periferní zařízení a jejich obsluha.

- Řadič (HW, vystavuje registry procesorů) vs. ovladač (SW, součást jádra, abstrahuje zařízení); BIOS/UEFI = inicializace a inventarizace zařízení při bootu.
- Speciální I/O instrukce (x86 in/out, oddělený I/O adresový prostor) vs. paměťově mapované I/O: registry na běžných adresách, přístup = load/store; deklarovat **volatile** (každý přístup skutečně provést, překladač nesmí slučovat ani přeskakovat).
- PIO s aktivním čekáním (polling): (1) zkontroluj stav, (2) nastav parametry (LBA, DMA adresa), (3) zapiš příkaz $\rightarrow$ `BUSY` = 1, (4) polluj `BUSY` bit, (5) přeber data. Mutex serializuje souběžný přístup; vždy uvolnit na *všech* cestách (i při chybě).
- Typy přerušení: **externí** (IRQ od HW, maskovatelná), **výjimka** (trap = po instrukci, fault = před $\rightarrow$ rollback + po obsluze zopakování), **softwarové** (int n).
- x86: 256 přerušení; 0–31 vyhrazeny (#0 dělení nulou, #xE výpadek stránky, #xD obecné porušení ochrany (general protection)); IDT = tabulka vektorů.
- HW reakce na přerušení: dokončí instrukci $\rightarrow$ zjistí číslo přerušení $\rightarrow$ indexuj IDT $\rightarrow$ ulož IP + příznaky $\rightarrow$ přepni do ring 0 $\rightarrow$ skok. SW (handler): ulož registry $\rightarrow$ obsluha (přebere data, doplní mapování, ...) $\rightarrow$ obnov $\rightarrow$ `iret` (obnov IP + příznaky + režim).
- DMA: řadič přenese blok dat přímo do fyzické paměti bez zatěžování procesoru; procesor nastaví FA cíle + příkaz; po dokončení přerušení.

Procesy, vlákna, plánování a soubory.

- Program (pasivní) → proces (instance: vlastní adresový prostor + prostředky) → vlákno (tok instrukcí: vlastní PC, zásobník, registry; sdílí kód, data, haldu, otevřené soubory). Vlákna jsou levná (sdílená paměť), ale vyžadují synchronizaci; procesy jsou izolované.
- Kontext vlákna: registry (PC, SP, příznaky). Přepnutí procesu navíc: přepni stránkovací tabulku ⇒ typicky invaliduj nebo přeznač (ASID) TLB.
- Kooperativní: vlákno se vzdá procesoru samo (yield/blokování na I/O). Preemptivní: hardwarový časovač → přerušeni → OS odebere procesor. Moderní OS: preemptivní.
- Plánování: FCFS (nepreempt., konvojový efekt); SJF (min. průměrná čekací doba při současném příchodu, vyžaduje znát délku); Round Robin (kvantum q , preempt., spravedlivé). Výpočty: obrátková = dokončení – příchod; čekací = obrátková – doba běhu.
- Stav: Created → Ready → Running ↔ Blocked → Terminated. Plánovač volí z Ready; Blocked čeká na prostředek (I/O, zámek).
- Adresový prostor procesu: kód | konstanty | stat. data | neinit. data | halda↓ | ... | zásobníky vláken↑.
- Soubor = abstraktní proud bajtů; rozhraní: open/read/write/seek/close; deskriptory 0 = stdin, 1 = stdout, 2 = stderr.

Virtuální paměť a překlad adres.

- VA (instrukce, ukazatele v kódu) vs. FA (sběrnice, paměťový řadič); překlad dělá MMU. Pravidlo: ukazatele v kódu = VA; DMA adresa předávaná řadiči = FA; čísla rámců v položkách stránkovací tabulky = FA.
- Stránka / rámec = 2^k B. VA = VPN || offset (dolních k bitů). Překlad jednoúrovňovou tabulkou: adresa PTE = báze + VPN-sizeof(PTE); FA = PFN- 2^k + offset.
- Rozměry (32-bit, 4 KiB stránky): $k = 12$, offset 12 b, VPN 20 b, 2^{20} položek po 4 B ⇒ tabulka 4 MiB/proces. Offset se nepřekládá, prochází beze změny.
- Příznaky PTE: P (present), W (writable), X (executable), A (accessed), D (dirty). Výpadek stránky = P = 0 nebo zakázaný přístup ⇒ výjimka #0xE → OS doplní mapování.
- TLB: asociativní cache (VPN → PFN); zásah = překlad okamžitě bez přístupu do paměti; minutí = průchod stránkovací tabulkou + uložení do TLB. ASID = identifikátor adresového prostoru: TLB může obsahovat překlady více procesů zároveň bez flushování při každém přepnutí.
- Ochrana: každý proces má vlastní tabulku ⇒ stejná VA dvou procesů vede na různé (nebo žádné) rámce ⇒ cizí paměť neviditelná. Sdílení záměrné: OS namapuje tytéž rámce do tabulek obou procesů.

Synchronizace a paralelismus.

- Race condition: výsledek závisí na prokládání vláken; typicky „přečti–uprav–zapiš“ bez atomicity (x++ jsou 3 instrukce). Diagnóza: najdi sdílená data + neatomický přístup bez zámku.
- Tři podmínky správného řešení: vzájemné vyloučení (v sekci nejvýše 1 vlákno), trvalý postup (bez deadlocku), omezené čekání (bez vyhladovění).
- Aktivní čekání (spin-lock, spotřebuje procesor; vhodné jen pro velmi krátká čekání) vs. pasivní čekání (blokující semafor/mutex; OS probudí, procesor dělá mezitím jiné věci).
- Semafor: čítač + fronta blokováných vláken; down (P): čítač > 0 → sniž; jinak blokuj. up (V): probud čekajícího ze fronty, nebo (je-li prázdná) zvýš čítač. Mutex = binární semafor (init 1). Obě operace atomické.
- Producent–konzument s kapacitou N : free (init N) + items (init 0) + mtx (init 1). Invariant: free + items = N . Pořadí: down (free/items) před down (mtx), jinak deadlock (vlákno by drželo mutex a čekalo na místo, které nikdo neuvolní).
- FIFO semafor mezi procesy: capacity (init N) + content (init 0); send: down (capacity), push, up (content); receive: down (content), pop, up (capacity).
- Vzájemná signalizace dvěma semaforey: vlákno A signalizuje do semB, B do semA ⇒ žádné vlákno nemůže vzít vlastní token; odstraní chybu jednosemaforové konstrukce (A si vezme zpět svůj token).

- Deadlock – Coffmanovy nutné podmínky: vzájemné vyloučení, drž a čekej, neodnímatelnost, cyklické čekání. Prevence: zamykat vždy ve stejném pevném pořadí.

Zdroje

- J. Yaghob, M. Kruliš: slidy přednášek předmětu NSWI170 *Počítačové systémy*, MFF UK (ročník 2024/25; primární zdroj části o operačních systémech: režimy procesoru, jádro, periferie, procesy a plánování, virtuální paměť, synchronizace).
<https://teaching.ms.mff.cuni.cz/nswi170-web/>
- L. Bulej, P. Ježek: slidy přednášek předmětu NSWI120 *Principy počítačů*, MFF UK (archivní sada 2011/12; reprezentace dat, instrukční sada, přerušení).
https://d3s.mff.cuni.cz/legacy/teaching/principles_of_computers/ar-20112012/
Stránka předmětu: <https://d3s.mff.cuni.cz/teaching/nswi120/>
- S. J. Matthews, T. Newhall, K. C. Webb: *Dive Into Systems*, verze 1.2, 2023 (volně dostupná učebnice; dvojkový doplněk, IEEE 754, překlad konstrukcí jazyka C na instrukce).
<https://diveintosystems.org/book/>
- MFF UK: požadavky ke státní závěrečné zkoušce (Informatika, Bc.) a zadání otázek z minulých termínů.
<https://www.mff.cuni.cz/cs/studenti/bakalarske-studium/statni-zaverecne-zkousky/bakalarske-statni-zkousky-studijniho-programu-informatika>