

Příklady otázek – Architektura počítačů a operačních systémů

Reálné otázky ze SZZ 2022–2026 (I4)

Jakub Klesá

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

Obsah

1	Virtuální paměť	1
2	Synchronizace	3
3	Stránkování	5
4	Překlad adresy	6
5	Architektura počítačů a operačních systémů	8
6	Ovladač pro řadič disku	9
7	Semafor	11
8	Jednoduchý správce paměti	13
9	Ukládání dat v binárním souboru	16
10	Semafor mezi procesy	18

Poznámka. Tento dokument obsahuje reálné otázky z minulých termínů SZZ (2022–2026) zařazené do okruhu I4. U otázek s obrázkem je grafika oříznutá z originálního PDF (viz odkaz na zdroj). Text, číselná data a tabulky jsou přepsané věrně. Zadání i oficiální náčrty řešení pocházejí ze sad zveřejněných na webu MFF UK:

<https://www.mff.cuni.cz/cs/studenti/bakalarske-studium/statni-zaverecne-zkousky/bakalarske-statni-zkousky-studijniho-programu-informatika>

Náčrty označené jako doplněné jsou vlastní, oficiální sada je k dané otázce neobsahuje. Vlastní doplnění k oficiálnímu náčrtu jsou zřetelně oddělena. Pojmy a značení odpovídají přednáškám předmětů NSWI170 *Počítačové systémy* (<https://teaching.ms.mff.cuni.cz/nswi170-web/>) a NSWI120 *Principy počítačů* (<https://d3s.mff.cuni.cz/teaching/nswi120/>).

1 Virtuální paměť

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

Zadání.

Uvažujme procesor s podporou stránkování. Virtuální i fyzická adresa má 32 bitů, používá se jednoúrovňová stránkovací tabulka a velikost stránky je 4 KiB. Vedle nezbytných atributů lze

pro každou stránku nastavit, zda ji smí proces modifikovat a zda v ní lze spouštět kód. Procesor navíc poskytuje informaci, zda se ke stránce přistupovalo a zda do ní bylo zapsáno.

Část A.

1. Navrhněte a schematicky znázorněte formát položky stránkovací tabulky a vysvětlete význam jednotlivých polí. Položka musí obsahovat všechny nutné informace a být efektivně přístupná (nejvýše 1 čtení z paměti).
2. Kolik položek bude mít stránkovací tabulka a kolik paměti zabere pro každý proces?

Část B.

1. Napište (v pseudokódu) funkci `create_pte` se signaturou `pte_t create_pte(address_t phys_addr, bool present, bool writable, bool executable)`, která pro zadanou fyzickou adresu a atributy vrátí položku stránkovací tabulky.
2. Napište (v pseudokódu) funkci `void set_mapping(pte_t[] page_table, address_t virt_addr, pte_t entry)`, která do tabulky pro zadanou virtuální adresu vloží záznam.

Část C. Souvislý blok 4 stránek virtuální paměti od adresy `0x0000B000` byl namapován na souvislý blok fyzické paměti. Blok je část heapu (lze číst a modifikovat, nelze v něm spouštět kód). Proces přečetl proměnnou (4 B) z virtuální adresy `0x0000CE90`, což bylo přeloženo na fyzickou adresu `0xA1018E90`. Pozměněnou hodnotu zapsal na virtuální adresu `0x0000D854`.

1. Na kterých fyzických adresách jsou uloženy položky stránkovací tabulky pro tento blok, je-li tabulka v souvislém bloku fyzické paměti od adresy `0x13400000`?
2. Uveďte hexadecimální hodnoty všech položek tabulky pro tento rozsah, včetně všech atributů.

V odpovědi důsledně uvádějte číselnou soustavu – prefix `0x` označuje šestnáctkovou soustavu.

Náčrt řešení.

Část A.2 (rozměry tabulky). Stránka má $4 \text{ KiB} = 2^{12} \text{ B}$, takže offset uvnitř stránky tvoří dolních 12 bitů a číslo stránky horních $32 - 12 = 20$ bitů. Stránkovací tabulka má proto $2^{20} = 1\,048\,576$ položek. Při velikosti položky 4 B (32 bitů) zabere tabulka $2^{20} \cdot 4 \text{ B} = 4 \text{ MiB}$ na proces.

Část A.1 (návrh položky – PTE, 32 bitů). Číslo fyzického rámce (PFN) má rovněž $32 - 12 = 20$ bitů, vejde se tedy do horních 20 bitů položky [31:12]. Dolních 12 bitů (které by u adresy tvořily offset) zbývá na příznaky – těch potřebujeme jen 5, takže 4-bytová položka bohatě stačí a celá se přečte jedním přístupem do paměti. Rozložení polí je volbou návrháře; v dalším pevně zafixujeme toto:

bity [31:12]	bit 6	bit 5	bit 2	bit 1	bit 0
PFN (číslo rámce)	D	A	X	W	P

kde **P** = Present (platnost záznamu), **W** = Writable (proces smí stránku modifikovat), **X** = eXecutable (lze v ní spouštět kód), **A** = Accessed (ke stránce se přistupovalo), **D** = Dirty (do stránky bylo zapsáno). Bity 3, 4, 7–11 zůstávají rezervované (zde 0). Bity A a D nastavuje hardware, ostatní spravuje OS.

Část B (pseudokód).

```
function create_pte(phys_addr, present, writable, executable):
    // PFN je v horních 20 bitech, dolní offsetové bity vynulujeme
    pte = phys_addr & 0xFFFFF000
    // příznaky vskládáme podle zvoleného rozložení z části A
    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):
    // cislo stranky = hornich 20 bitu virtualni adresy
    index = virt_addr >> 12
    page_table[index] = entry

```

Část C (překlad a položky tabulky). Stránka má 0x1000 B, číslo stránky je horních 20 bitů adresy (tj. adresa >> 12), offset dolních 12 bitů.

Čtená virtuální adresa 0x0000CE90: číslo stránky = 0xC, offset = 0xE90. Fyzická adresa 0xA1018E90 má stejný offset 0xE90 (souhlasí) a rámec = 0xA1018. Blok 4 stránek od 0x0000B000 jsou virtuální stránky 0xB, 0xC, 0xD, 0xE. Protože je mapování souvislé, rámce navazují:

0xB → 0xA1017, 0xC → 0xA1018, 0xD → 0xA1019, 0xE → 0xA101A.

Příznaky A a D: zápis proběhl na 0x0000D854 (stránka 0xD) ⇒ stránka 0xD je Accessed i Dirty. Čtení proběhlo ze stránky 0xC ⇒ stránka 0xC je jen Accessed (A=1, D=0). Stránky 0xB a 0xE nebyly použity (A=0, D=0). Protože jde o heap, je u všech P=1, W=1, X=0.

C.1 – adresy položek. Tabulka začíná na 0x13400000, položka má 4 B, adresa položky = báze + číslo_stránky · 4:

0xB → 0x1340002C, 0xC → 0x13400030, 0xD → 0x13400034, 0xE → 0x13400038.

C.2 – hodnoty položek. Hodnota = (PFN << 12) | příznaky. Příznakové byty dle zvoleného rozložení: 0x03 = P+W (bity 0,1); 0x23 = P+W+Accessed (přidán bit 5 = 0x20); 0x63 = P+W+A+Dirty (navíc bit 6 = 0x40).

stránka	rámec (PFN)	A, D	položka
0xB	0xA1017	A=0, D=0	0xA1017003
0xC	0xA1018	A=1, D=0	0xA1018023
0xD	0xA1019	A=1, D=1	0xA1019063
0xE	0xA101A	A=0, D=0	0xA101A003

Přesné hodnoty příznakových bitů závisejí na zvoleném rozložení položky PTE z části A; při jiném rozložení polí by se příznakové byty odpovídajícím způsobem změnily (PFN a adresy položek zůstávají stejné).

(V PDF bez náčrtu řešení; náčrt doplněn k výkladu okruhu, číselné výsledky ověřeny numericky.)

2 Synchronizace

(Reálná otázka: 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 popište sémantiku jeho operací.

2. Pomocí rozhraní z prvního bodu implementujte datovou strukturu a funkce pro řešení problému producent-konzument, jejíž skica je uvedena níže. Není potřeba implementovat vlastní datovou strukturu pro udržení seznamu vyprodukovaných (ale nespotřebovaných) dat, stačí popsat její rozhraní. To ale samo o sobě nesmí poskytovat žádné záruky ohledně synchronizace.

```
typedef struct {  
    ...  
} prodcons_t;  
  
void prodcons_init (prodcons_t *prodcons, int capacity) { ... }  
  
void prodcons_produce (prodcons_t *prodcons, void *item) { ... }  
void *prodcons_consume (prodcons_t *prodcons) { ... }
```

V odpovědi používejte programovací jazyk C nebo C++, ale kromě rozhraní z prvního bodu nepoužívejte žádné další nástroje pro synchronizaci.

Náčrt řešení.

(1) Rozhraní semaforu. Semafor drží nezáporný čítač a frontu čekajících vláken. Operace down (P): je-li čítač > 0 , sníží ho; jinak vlákno *pasivně* zablokuje a zařadí do fronty. Operace up (V): je-li někdo ve frontě, probudí ho; jinak čítač zvýší. Obě operace jsou atomické.

(2) Producent–konzument. Použijeme tři semafore: **free** (volná místa, init *capacity*), **items** (připravené položky, init 0) a binární **mtx** (vzájemné vyloučení nad bufferem, init 1). Buffer **buf** má jen rozhraní **buffer_push**/**buffer_pop** bez vlastní synchronizace.

```
typedef struct {  
    semaphore free; // init = capacity (volna mista)  
    semaphore items; // init = 0 (pripravene položky)  
    semaphore mtx; // init = 1 (vzajemne vyloučení nad bufferem)  
    buffer_t buf; // push/pop bez vlastní synchronizace  
} prodcons_t;  
  
void prodcons_produce(prodcons_t *pc, void *item) {  
    down(&pc->free); // pockej na volne místo  
    down(&pc->mtx);  
    buffer_push(&pc->buf, item);  
    up(&pc->mtx);  
    up(&pc->items); // ohlas novou položku  
}  
  
void *prodcons_consume(prodcons_t *pc) {  
    down(&pc->items); // pockej na položku  
    down(&pc->mtx);  
    void *item = buffer_pop(&pc->buf);  
    up(&pc->mtx);  
    up(&pc->free); // uvolni místo  
    return item;  
}
```

Invarianty **free** = *capacity* – počet položek a **items** = počet položek zaručují, že se nezapisuje do plného a nečte z prázdného bufferu. Počítací semafor se získává *před* **mtx** (jinak by hrozil deadlock). **mtx** je potřeba i při jediném producentovi a konzumentovi, protože **push** a **pop** mohou běžet současně nad touž nesynchronizovanou strukturou.

(Náčrt doplněn k výkladu okruhu; otázka nemá v PDF řešení, správnost ověřena rozborem invariantů a pořadí operací.)

3 Stránkování

(Reálná otázka: 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.

Předpokládejte procesor s virtuálními a fyzickými adresami velikosti 4 B, který pro překlad virtuálních adres používá softwarově plněný TLB s pevnou velikostí stránky 4 MB.

Položka v TLB má velikost 4 B a obsahuje (po řadě od nejvyšších bitů) potřebnou část virtuální a fyzické adresy, operačním systémem přiřazený ASID (identifikátor adresového prostoru) a bity dirty a valid. Zbývající dva nejnižší bity jsou rezervované jako 0.

1. Načrtněte formát položky TLB včetně počtu bitů jednotlivých polí.
2. Předpokládejte následující obsah TLB:

0x35C0C124
0x35C0613C
0x35C08120
0x36005134
0x36009128
0x3600D12C
0x3640E124
0x3640A120
0x3640413C

Předpokládejte, že aktuální ASID je nastaven na hodnotu 0x12 a dále se nemění.

Co se stane při přístupech na následující virtuální adresy? V případě, že dojde k úspěšnému překladu, napište konkrétní fyzickou adresu, ke které procesor přistoupí. V případě, že překlad nebude moci procesor provést, ve stručnosti popište činnost procesoru a případnou (typickou) reakci operačního systému pro napravení situace.

Jednotlivé přístupy uvažujte izolovaně, tedy vycházejte vždy z obsahu TLB uvedeného výše, i kdyby v některém kroku případná reakce operačního systému mohla vést ke změně TLB.

- zápis na 0x35803580
- čtení z 0x35CD35CD
- čtení z 0x36003600
- zápis na 0x36403640

Odpovědi přiměřeně komentujte tak, aby bylo možné vidět, jak jste dospěli k výsledku.

Náčrt řešení.

(1) Formát položky. Stránka má $4\text{ MB} = 2^{22}\text{ B}$, takže offset je 22 bitů a číslo stránky i rámce má $32 - 22 = 10$ bitů. Položka (32 b) je od nejvyšších bitů: VPN [31:22] (10 b), PFN [21:12] (10 b), ASID [11:4] ($32 - 10 - 10 - 1 - 1 - 2 = 8$ b), *dirty* [3], *valid* [2], rezervované [1:0]. (Např. 0x35C0C124 → VPN 0x0D7, PFN 0x00C, ASID 0x12, $D=0$, $V=1$.)

(2) Přístupy. U každé adresy je VPN horních 10 bitů; položka platí, jen když má *shodné* VPN, $V=1$ a *shodný* ASID 0x12. Fyzická adresa = $\text{PFN} \cdot 2^{22} + \text{offset}$, offset jsou dolní 22 bity.

- 0x35803580 (zápis): VPN 0x0D6 – v TLB *není* ⇒ **minutí**. Softwarově plněný TLB: procesor vyvolá výjimku TLB miss, obslužná rutina OS dohledá překlad ve stránkovací tabulce a doplní TLB (a instrukci zopakuje), případně vyvolá výpadek stránky.

- 0x35CD35CD (čtení): VPN 0x0D7; platná shoda je 0x35C0C124 (PFN 0x00C; ostatní mají ASID 0x13 nebo $V=0$). FA = 0x030D35CD.
- 0x36003600 (čtení): VPN 0x0D8; platná shoda 0x3600D12C (PFN 0x00D; 0x36009128 má $V=0$). FA = 0x03403600.
- 0x36403640 (zápis): VPN 0x0D9; platná shoda 0x3640E124 (PFN 0x00E, $D=0$), překlad vede na FA = 0x03803640. Jde ale o zápis na stránku s $D=0$: zápis hned neproběhne, procesor vyvolá výjimku (u MIPS „TLB modified“). OS ověří, že stránka je zapisovatelná, nastaví $D=1$ a instrukci zopakuje; zápis pak proběhne na uvedenou FA. Není-li stránka zapisovatelná, jde o porušení ochrany.

(Náčrt doplněn k výkladu okruhu; otázka nemá v PDF řešení, dekodování i překlady ověřeny skriptem `verify_szz.py`.)

4 Překlad adresy

(Reálná otázka: 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.

Předpokládejme hypotetický procesor s pamětovou architekturou, kde virtuální i fyzický prostor používá 16-bitové adresování. Pro překlad adres se používá 1-úrovňové stránkování s 64 B stránkami. Záznamy ve stránkovací tabulce (které mají také 16 bitů) využívají offsetové bity pro přidružené příznaky, nejméně významný bit je použit jako příznak platnosti (tj. 1 = záznam je platný, 0 = záznam není platný).

Stránkovací tabulka, jejíž výpis následuje níže, se nachází na adrese 0x1d40. Výpis zobrazuje pro jednoduchost přímo 16-bitové položky stránkovací tabulky, tedy není nutné řešit pořadí bytů v reprezentaci čísel.

	+0x0	+0x2	+0x4	+0x6	+0x8	+0xA	+0xC	+0xE
0x1d40	0xdecd	0x0149	0x21c9	0x2b19	0x8f85	0x65d9	0x68a1	0x2151
0x1d50	0x2fb1	0x8249	0xabc5	0x3219	0xeacd	0x2651	0x9111	0x4249
0x1d60	0x5085	0x8e4d	0xe731	0xb109	0xb751	0x7fe2	0x3d6a	0xcf24
0x1d70	0x9c9c	0x6011	0x3d11	0xd589	0xc14d	0x748d	0xa019	0x41c9
0x1d80	0x0320	0x0e2a	0xd2d6	0xb0a8	0x8b44	0x3aa2	0xe4ce	0xcde0
0x1d90	0x5068	0xfbfb	0x427e	0x007c	0x7bba	0x578e	0x5770	0x4f76
0x1da0	0x85b6	0xe920	0x7768	0xcaf0	0xe3c0	0x76fa	0xc84a	0xdc66
0x1db0	0x94d4	0x3fb2	0xb986	0xe52c	0x1ddc	0x957e	0x8ccc	0x82a2
0x1dc0	0x4f3c	0xf6d2	0x8fc2	0xee90	0x9792	0xa43a	0xc590	0x9180
0x1dd0	0x1386	0x2b92	0xdcf0	0x62a0	0x47d8	0x9ede	0x9614	0x8d9c
0x1de0	0xda10	0x16d0	0xc68c	0xa108	0x45c8	0x1f60	0x4a18	0xe9ce
0x1df0	0x4c4f	0xbbe9	0x9535	0x74eb	0x6b19	0x7aad	0x36b1	0x85fd
0x1e00	0xcfe3	0x7dd3	0x4917	0x54ad	0x9391	0x92c5	0x0ab9	0x9fa7
0x1e10	0xd390	0xddfa	0x6330	0x33a2	0xad22	0xabbe	0xe21a	0x04dc
0x1e20	0x43f8	0x4188	0xfe3c	0xba68	0xb082	0x2040	0x126e	0xdecc
0x1e30	0xc206	0xb9a2	0xbe10	0x0cf8	0x81c8	0x564c	0x8140	0x923a
0x1e40	0x5bda	0xbd12	0xb794	0x910c	0x05f2	0x7636	0x9bd8	0xdede
0x1e50	0x85f2	0x21ca	0xf3c8	0x1bc2	0xde4e	0xa87c	0x9a5e	0xc348
0x1e60	0x80e0	0x37ea	0x39a8	0x7ef6	0xd8ca	0x79f4	0x286c	0x7f00
0x1e70	0xb046	0x0fe8	0x5256	0x72e8	0xe0cc	0xf6aa	0x39d8	0x8c4e
0x1e80	0xdd44	0x5088	0x789a	0x68d0	0x1748	0xfbb0	0x9130	0x8e9e
0x1e90	0x13bf	0xe44f	0x525f	0x8715	0x5389	0x849f	0xaeel	0xd451
0x1ea0	0x584a	0xe278	0x6776	0x32b0	0x82b0	0x8b1c	0x704c	0x2e8a
0x1eb0	0xabce	0x4020	0xd434	0xaea	0x6548	0xdf10	0x3ad8	0x388a
0x1ec0	0xb134	0x8d2c	0xe422	0x6ec8	0x101d	0xa595	0x0b55	0x7cf1
0x1ed0	0x69b9	0x431b	0xfbfb	0x293d	0x0075	0xd8d7	0x9059	0x1569
0x1ee0	0x4c73	0x8577	0x6efb	0x2e09	0xa58d	0xd858	0x39d2	0x00d8
0x1ef0	0x5eb8	0x92e6	0x5798	0x2314	0xb82c	0x34c2	0xbe16	0xbed6
0x1f00	0x4b12	0x4660	0xb082	0x2eee	0x3ee4	0x9140	0x4b42	0xef9e
0x1f10	0x8318	0x5a74	0x7aa2	0xda8a	0x1504	0x5cf4	0x9876	0x3632
0x1f20	0xb5be	0x802e	0xab94	0x0c46	0xd588	0xb138	0xc378	0x5a8a
0x1f30	0x94ea	0x1254	0x0bac	0xae2	0x2c6a	0xc5de	0x5b12	0xca64

Přeložte virtuální adresu 0x328f (0b0011001010001111 binárně) na fyzickou adresu:

1. Napište číslo stránky a offset pro danou virtuální adresu.
2. Napište fyzickou adresu položky ve stránkovací tabulce, která odpovídá zadané virtuální adrese.
3. Napište výslednou fyzickou adresu, pokud existuje překlad. Jinak uveďte, že neexistuje.

V odpovědi důsledně uvádějte číselnou soustavu – čísla bez prefixu jsou v desítkové soustavě, prefix 0x označuje šestnáctkovou soustavu, prefix 0b označuje dvojkovou soustavu. (To, jakou soustavu použijete, se ovšem samo o sobě nehodnotí.)

Náčrt řešení.

1. Stránka má $64 = 2^6$ B, takže offset jsou dolní 6 bitů a číslo stránky horních 10 bitů:

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

$$\text{číslo stránky} = \boxed{0xCA = 202}, \text{ offset} = \boxed{0x0F = 15}.$$

2. Položky tabulky jsou 16-bitové (2 B), položka stránky 202 je na adrese $0x1d40 + 202 \cdot 2 = 0x1d40 + 0x194 = \boxed{0x1ED4}$.
3. Na adrese $0x1ED4$ je ve výpisu hodnota $0xFBF7$. Nejnižší bit je 1, položka je *platná*. Číslo rámce tvoří horních 10 bitů (dolních 6 jsou příznaky): $0xFBF7 \& 0xFFC0 = 0xFBC0$. Fyzická adresa vznikne spojením rámce s offsetem: $0xFBC0 | 0x0F = \boxed{0xFBCF}$.

(Náčrt doplněn k výkladu okruhu; otázka nemá v PDF řešení, výpočet ověřen skriptem *verify_szz.py*.)

5 Architektura počítačů a operačních systémů

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

Zadání.

Mějme následující třídu v pseudokódu:

```

1 class LimitedStack {
2 public:
3     LimitedStack(int md) {
4         currentDepth = 0;
5         maxDepth = md;
6         stack = new int[maxDepth];
7     }
8
9     int getDepth() {
10         return currentDepth;
11     }
12
13     bool pushValue(int v) {
14         if (currentDepth >= maxDepth)
15             return false;
16         stack[currentDepth] = v;
17         currentDepth++;
18         return true;
19     }
20
21 private:
22     int maxDepth;
23     int currentDepth;
24     int[] stack; // int *stack; pro C++
25 };

```

Dále předpokládejme, že máme multiprocessorový systém a program si spustí několik vláken, která budou mít všechna přístup ke sdílené proměnné typu `LimitedStack`. Tato vlákna budou volat podle své potřeby funkce `getDepth` a `pushValue`.

1. Je možné, že v programu nastane jev zvaný *race condition*? Pokud ano, napište rozsahy řádek nebo vyznačte přímo v uvedeném kódu řádky, které představují kritickou sekci.
2. Je možné, aby nějakým způsobem nastala situace, kdy při volání funkce `pushValue` přeteče zásobník (program se bude pokoušet zapisovat mimo alokované pole)? Zdůvodněte.
3. Pokud se domníváte, že v uvedeném kódu může nastat *race condition*, pokuste se tento problém odstranit úpravou programu (včetně možného přidání nějakých nových řádek).

Kde je to důležité, ve své odpovědi uvažujte jazyk C#, C++ nebo Java (a vaši volbu vyznačte).

Náčrt řešení.

1. Ano, 10 a 14-17.
2. Ano, T1 i T2 vykonají řádek 14 souběžně. Přečtou shodnou proměnnou `currentDepth`, která má hodnotu `maxDepth - 1`, takže test selže. Následně po testu je T2 plánovačem zastaveno, T1 funkci dokončí a zvětší `currentDepth`. Poté je T2 znovu naplánováno a při zápisu hodnoty použije zvětšený index sahající za pole.
3. Záleží na jazyku. V Javě triviálně použitím `synchronized` metod. C# buď použije `lock` na nějaký vedlejší objekt nebo se použije `[MethodImpl(MethodImplOptions.Synchronized)]`. C++ použije třeba `Mutex` a buď explicitní `lock` a `unlock` nebo lépe `lock_guard`. Pozor na odemknutí před každým `return`.

Doplňující vysvětlení (nad rámec oficiálního náčrtu): Řádek 10 patří do kritické sekce proto, že čte `currentDepth`, kterou jiné vlákno právě mění; řádky 14–17 čtou i zapisují `currentDepth` a zapisují do `stack`. Samotné `currentDepth++` navíc není atomické (načti, přičti, ulož), takže se dvě souběžná zvýšení mohou slít v jedno. Zámek musí obepnout *celý* úsek od testu na řádku 14 po inkrementaci: kdyby chránil jen zápis, dvě vlákna by dál mohla projít testem se stejnou hodnotou. Možná úprava v C++:

```
std::mutex mtx; // nova clenska promenna

int getDepth() {
    std::lock_guard<std::mutex> g(mtx);
    return currentDepth;
}

bool pushValue(int v) {
    std::lock_guard<std::mutex> g(mtx); // odemkne se pri kazdem return
    if (currentDepth >= maxDepth)
        return false;
    stack[currentDepth] = v;
    currentDepth++;
    return true;
}
```

6 Ovladač pro řadič disku

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

Zadání.

Napište implementaci funkce ovladače disku pro načtení jednoho bloku.

Uvažovaný disk je řízen pěti pamětově mapovanými registry, které slouží k zápisu příkazů (a jejich parametrů) a ke čtení aktuálního stavu zařízení (uvažujeme zařízení bez podpory přerušení, s obsluhou s aktivním čekáním).

Offset	Registr	Typ	Popis registru
0	Status	R	Bitové pole pro aktuální stav řadiče (pouze pro čtení, zápisy jsou ignorovány).
4	Size	R	Velikost disku v blocích (pouze pro čtení, zápisy jsou ignorovány).
8	Command	W	Zápis do registru spustí zadaný příkaz (1 čtení, 2 zápis).
12	LBA	W	Logická adresa bloku pro příkaz zadaný v <i>Command</i> .
16	DMA	W	Fyzická adresa paměti pro uložení načtených dat z disku.

Registr Status (offset 0) je bitové pole:

31	29	...	2	1	0
0	0	...	0	BUSY	ERR

BUSY – Bit je nastaven po zadání příkazu a vynulován po dokončení operace.

ERR – Bit signalizující chybový stav (1 znamená chybu).

Bloky jsou na disku adresovány obvyklým způsobem pomocí LBA, jednotlivé bloky mají pevně danou velikost 512 bajtů. Přenos čtených či zapisovaných dat probíhá pomocí DMA (tedy z pohledu této otázky není třeba řešit nic víc než nastavení fyzické adresy pro uložení dat, vlastní čtení z disku a zápis do paměti je plně v režii řadiče), dokončení operace je signalizováno stavovým registrem (předpokládáme 32-bitový systém, velikost disku je omezena na 2 TB).

Dopíšte těla dvou níže uvedených funkcí a navrhnete strukturu pro uchování informací o disku (předpokládáme, že k systému může být připojeno více disků a ty jsou na úrovni jádra operačního systému rozlišeny různými instancemi struktury `disk_t`). Obě funkce vrací `true` pokud se daná operace zdařila, jinak `false` (bez dalších detailů).

Funkce `disk_init` je volána jednou při inicializaci daného zařízení, parametr `register_address` je virtuální adresa mapovaných registrů (tj. přímo adresa stavového registru). Funkce `disk_read_block_waiting` přečte jeden blok z disku. Načtený blok má být zapsán na fyzickou adresu `data_phys_addr` (váš kód může předpokládat její správnost). Návrat z funkce musí proběhnout až po dokončení čtení (je možné využít aktivní čekání).

Vaše implementace musí alespoň triviálním způsobem ošetřit možný současný přístup k disku z více procesů.

```
typedef struct { ... } disk_t;

bool disk_init(disk_t *disk, uint32_t register_address) { ... }

bool disk_read_block_waiting(disk_t *disk, size_t lba, uint32_t data_phys_addr) { ... }
```

Náčrt řešení.

Nástin zdrojového kódu (bez symbolických konstant a komentářů):

```
typedef volatile struct {
    uint32_t status;
    uint32_t size;
    uint32_t command;
    uint32_t lba;
    uint32_t dma;
} disk_regs_t;

typedef struct {
    size_t max_lba;
    disk_regs_t *ctl;
    mutex_t mutex;
} disk_t;

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)) {
        if (!disk_is_ok(disk)) {
            return false;
        }
    }
    return true;
}
```

```

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);
}

bool disk_read_block_waiting(disk_t *disk, size_t lba, uint32_t data_phys_addr) {
    if (lba >= disk->max_lba) {
        return false;
    }
    mutex_lock(&disk->mutex);
    bool ok = disk_wait_for_ready(disk);
    if (!ok) {
        mutex_unlock(&disk->mutex);
        return false;
    }

    disk->ctl->lba = lba;
    disk->ctl->dma = data_phys_addr;
    disk->ctl->command = 1;

    ok = disk_wait_for_ready(disk);
    mutex_unlock(&disk->mutex);

    return ok;
}

```

Doplňující vysvětlení (nad rámec oficiálního náčrtu): Ve funkcích `disk_is_ok` a `disk_is_ready` chybí v náčrtu závorky. V C má `==` vyšší prioritu než bitové `&`, takže `status & 1 == 0` znamená `status & (1 == 0)`, tj. `status & 0`, a to je vždy 0. Správně má být `(status & 1) == 0` a `(status & 2) == 0`. Zbytek náčrtu: struktura registrů je `volatile`, aby překladač čtení stavu v čekací smyčce pokaždé skutečně provedl a zápisy parametrů a příkazu nepřeházel. Mutex serializuje přístup více procesů k jediné sadě registrů a odemyká se na všech návratových cestách. Parametry (LBA, DMA) se zapisují *před* příkazem, protože zápis do `Command` operaci rovnou spouští.

7 Semafor

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

Zadání.

Knihovna `System.Threading` jazyka C# obsahuje třídu `Semaphore` s metodami odpovídajícími operacím klasického semaforu, *P* (`WaitOne`) a *V* (`Release`). Objekt vytvořený pomocí `new Semaphore(0, 1)` odpovídá binárnímu semaforu inicializovanému nulou. Pokud více než jedno vlákno čeká na tentýž semafor, pořadí, ve kterém budou vlákna spouštěna po uvolnění semaforu, není specifikováno.

Program vpravo má produkovat výstup `OneTwoThreeFourFive`, přitom liché části výstupu mají být vypisovány funkcí `f1` a sudé části funkcí `f2`, přičemž tyto funkce běží v různých vláknech.

Synchronizace, implementovaná pomocí semaforu, téměř funguje, ale obsahuje časově závislé chyby (race conditions).

```

1 class Program
2 {
3     private static Semaphore sem;

```

```

4
5 private static void f1()
6 {
7     Console.Write("One");
8     sem.Release();
9     sem.WaitOne();
10    Console.Write("Three");
11    sem.Release();
12    sem.WaitOne();
13    Console.Write("Five");
14 }
15
16 private static void f2()
17 {
18     sem.WaitOne();
19     Console.Write("Two");
20     sem.Release();
21     sem.WaitOne();
22     Console.Write("Four");
23     sem.Release();
24 }
25
26 static void Main(string[] args)
27 {
28     sem = new Semaphore(0, 1);
29     Thread t1 = new Thread(f1);
30     Thread t2 = new Thread(f2);
31     t1.Start();
32     t2.Start();
33     t1.Join();
34     t2.Join();
35 }
36 }

```

1. Popište scénář, který vede k jinému výstupu než **OneTwoThreeFourFive**, nebo končí uváznutím (deadlock). Scénář zapište jako posloupnost čísel řádek, přičemž každé číslo reprezentuje *ukončení* příkazu na dané řádce. Pouze vnitřky funkcí **f1** a **f2** jsou relevantní.
2. Existovaly by v tomto kódu časově závislé chyby i v případě, že by knihovna **C#** garantovala FIFO pořadí spouštění vláken čekajících ve **WaitOne**?
3. Napište řešení, které neobsahuje časově závislé chyby a spolehlivě vypisuje požadovaný výstup **OneTwoThreeFourFive**. Použijte přitom dva semafore a pouze jejich funkce **Release()** a **WaitOne()**. Řešení nesmí používat žádné jiné proměnné nebo objekty sdílené mezi vlákny kromě těchto dvou semaforů. Volání **Console.Write** musejí samozřejmě zůstat tam, kde jsou.

(Jazyk **C#** je v této otázce použitý pouze jako generický zástupce běžných programovacích jazyků, otázka ani řešení s jazykem **C#** jako takovým nesouvisí.)

Náčrt řešení.

1. Všechny scénáře, v lexikografickém pořadí:
 - (a) 7-8-9-10-11-12-13-deadlock(18+34): **OneThreeFive**
 - (b) 7-8-9-10-11-18-19-20-12-13-deadlock(21+34): **OneThreeTwoFive**

- (c) 7-8-9-10-11-18-19-20-21-22-23-12-13: *OneThreeTwoFourFive*
- (d) 7-8-18-19-20-9-10-11-12-13-deadlock(21+34): *OneTwoThreeFive*
- (e) ~~7-8-18-19-20-9-10-11-21-22-23-12-13: *OneTwoThreeFourFive*~~ – *The expected program behavior*
- (f) 7-8-18-19-20-21-22-23-9-10-11-12-13: *OneTwoFourThreeFive*

Postačující odpovědí je kterýkoli z výše uvedených scénářů kromě (e).

2. Ano. Všechny výše uvedené scénáře fungují i pro FIFO čekání.
3. Viz kód níže:

```
private static void f1()
{
    Console.Write("One");
    semA.Release();
    semB.WaitOne();
    Console.Write("Three");
    semA.Release();
    semB.WaitOne();
    Console.Write("Five");
}

private static void f2()
{
    semA.WaitOne();
    Console.Write("Two");
    semB.Release();
    semA.WaitOne();
    Console.Write("Four");
    semB.Release();
}
```

Doplňující vysvětlení (nad rámec oficiálního náčrtu): Kořen chyby je v tom, že jediný semafor nerozlišuje, *komu* je signál určen: **f1** po **Release** (ř. 8) pokračuje hned **WaitOne** (ř. 9) a může si svůj vlastní token vzít dřív, než **f2** vůbec dojde na ř. 18. Proto nepomůže ani FIFO pořadí probouzení: ve scénáři (a) na semaforu v tu chvíli nikdo nečeká, takže není koho probouzet ve frontě. Se dvěma semaforů **f1** signalizuje jen do **semA** a čeká jen na **semB**, **f2** naopak; žádné vlákno tak nemůže spotřebovat token, který samo vydalo, a výpisy se střídají v pevném pořadí.

8 Jednoduchý správce paměti

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

Zadání.

Předpokládejte v celé otázce pouze kontext 32-bitového **little endian** procesoru, **bitem 0 rozumíme LSb**.

Naším cílem je v souvislém bloku paměti implementovat jednoduchý heap pro dynamickou alokaci podbloků paměti (neplést s haldou jako datovou strukturou). Celý náš heap bude mít pevnou velikost danou při inicializaci a nebude používat žádné optimalizace nad rámec zde popsaného. Paměť našeho heapu si chceme organizovat jako souvislou posloupnost podbloků, kde každý podblok má být vždy *zarovnaný na 2 byty*. Každý podblok heapu je buď *volný* nebo *obsazený* (naalokovaný).

Volné podbloky mají následující strukturu:

+0: Size [16-bit] = velikost podbloku **včetně této hlavičky**. Jelikož z výše uvedeného vyplývá, že bit 0 této položky by byl vždy 0, tak tuto nulu neukládáme. Místo toho pro nás bude mít bit 0 položky Size speciální význam – bude označovat, zda je podblok volný nebo obsazený. 0 = *volný*, 1 = *obsazený*. Tedy pro volný podblok:

bit 0: 0 = volný

+2: Next [16-bit] = offset dalšího volného podbloku od začátku celého heapu – tedy volné podbloky tvoří vlastně jednosměrně vázaný seznam. Poslední volný podblok má v této položce vyplněnou hodnotu -1.

+4: zbytek podbloku je nevyužitý a je vyplněn nulovými byty.

Obsazené podbloky mají následující strukturu:

+0: Size [16-bit] = má stejný význam jako u volného podbloku, s tím rozdílem, že zde:

bit 0: 1 = obsazený

+2: Payload – samotná data podbloku (náš alokátor vrací všechny tyto byty vynulované).

Pozor: obsazené podbloky tedy nemají položku Next – místo v paměti, které u volného podbloku náleželo položce Next, se alokací podbloku stane součástí Payloadu a musí být alokátozem na haldě vynulováno.

Pro celý heap si pamatujeme offset prvního volného podbloku od začátku celého heapu. Na počátku se heap skládá právě z 1 volného podbloku, jehož velikost odpovídá velikosti celého heapu. Při požadavku na alokaci nového podbloku dostaneme požadovanou velikost payloadu v bytech a použijeme první dostatečně velký volný podblok na nejnižším offsetu od začátku heapu (tj. strategie first-fit).

Předpokládejte následující kostru třídy v C#, kterou budete doplňovat (požadovanou implementaci můžete napsat v jazyce C# nebo Java nebo C++ – pro Javu nebo C++ předpokládejte ekvivalentní implementaci kostry v daném jazyce s tím, že parametry konstruktoru a metod si můžete vhodně upravit dle zvyklostí v daném jazyce) (**ushort** je 16-bitové bezznaménkové celé číslo):

```
class Heap {
    private ushort _firstFreeOffset = 0;
    private byte[] _heap;

    public Heap(byte[] availableMemory) {
        _heap = availableMemory;
        // TODO
    }
    private ushort FindFirstFree(ushort payloadSize) { // TODO }
    private void Mark(ushort offset, bool isFree) { // TODO }

    public void SetUshort(ushort offset, ushort value) { ... }
    public ushort GetUshort(ushort offset) { ... }
    public ushort Alloc(ushort payloadSize) { ... }
    public void Free(ushort payloadOffset) { ... }
}
```

Metody **SetUshort** a **GetUshort** **neprogramujte**, ale naopak je vhodně využijte ve svém kódu. Metoda **SetUshort** na offsetu od začátku heapu uloží **value** v little endian pořadí. Metoda **GetUshort** vrátí 16-bitovou hodnotu uloženou na heapu od daného **offsetu** (byty na heapu chápe v little endian pořadí).

Metody **Alloc** a **Free** **neprogramujte** – tyto metody bude programovat někdo jiný s využitím vašich metod. Pro úkol 4 níže se hodí vědět: metoda **Alloc** naalokuje na heapu místo pro **payloadSize** velký payload a vrátí offset pro payload; metoda **Free** bere offset payloadu vráceného nějakým voláním metody **Alloc** a podblok, do kterého payload náleží, uvolní.

Úkoly:

1. Doplníte implementaci konstruktoru, který inicializuje prázdný heap.
2. Napište implementaci metody **FindFirstFree**, která vrátí offset prvního volného podbloku vhodné velikosti. Stav podbloku tato funkce nijak nemění.
3. Napište implementaci metody **Mark**, která očekává, že na **offsetu** je platný podblok haldy, a změni jeho stav na „obsazený“ nebo „volný“ dle parametru **isFree**. Funkce nebude manipulovat s položkou **Next**.
4. Předpokládejte, že jsme pro náš heap dostali k dispozici 22 bytů – tato paměť je předem vyplněna nulovými byty. Předpokládejte, že provedeme s heapem následující operace:

```
A = Alloc(2)
B = Alloc(4)
C = Alloc(2)
Free(B)
```

Napište hexdump 22 bytů paměti konečného stavu heapu po provedení všech těchto operací – řešení vyplňte do jedné z níže uvedených tabulek (je jich zde redundantně více, kdybyste své řešení potřebovali opravit).

offset	+0	+1	+2	+3	+4	+5	+6	+7
0x0000								
0x0008								
0x0010								

Náčrt řešení.

Konečný stav heapu (hexdump 22 bytů):

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	–	–

Doplňující vysvětlení (nad rámec oficiálního náčrtu): Oficiální náčrt uvádí jen výsledný hexdump (úkol 4). Úkoly 1–3 v C#: potřebná velikost podbloku je payload + 2 (hlavička), zaokrouhleno nahoru na sudé číslo.

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

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

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

Odvození hexdumpu (22 B, na počátku jeden volný podblok @0 velikosti 22): Alloc(2) potřebuje 2+2 = 4 B, vezme @0 a zbytek @4 (18 B) zůstane volný; Alloc(4) potřebuje 6 B, vezme @4, volný zbytek @10 (12 B); Alloc(2) vezme @10 (4 B), volný zbytek @14 (8 B). Free(B) dostane offset payloadu 6, podblok je tedy @4; stane se volným a zařadí se do seznamu před @14. Výsledek: A @0 se Size = 4|1 = 5, volný @4 se Size 6 a Next = 14 = 0x0E, C @10 se Size 5, volný @14 se Size 8 a Next = 0xFFFF. Všechny 16-bitové hodnoty jsou little-endian (05 00, 0E 00, ...), payloady i zbytky volných podbloků jsou nulové.

9 Ukládání dat v binárním souboru

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

Zadání.

Aplikace ukládá svá data do binárního souboru, což je ilustrováno následujícím (částečným) hexadecimálním výpisem:

```
0000: 00 06 48 68 53 53 4C 4C 00 22 03 E8 FC 18 00 05 ..Hh SSLL .".. ....
0010: 48 65 6C 6C 6F 00 05 57 6F 72 6C 64 00 00 00 00 Hell o..W orld ....
0020: 00 00 00 2C 00 00 00 00 00 00 12 34 00 88 01 F4 ..., .... ...4 ....
```

Logicky soubor obsahuje *uzly*. Každý uzel obsahuje stejnou sadu *atributů*. Typy atributů jsou určeny *hlavičkou* souboru umístěnou na jeho začátku, která se skládá ze dvou bajtů udávajících počet atributů (v našem příkladu 6), následovaných typy atributů, kódovanými jedním bajtem pro každý atribut, odpovídající tomuto výčtu:

```
enum AttrType { TUInt16 = 0x48, TLink = 0x4C, TString = 0x53, TSInt16 = 0x68};
```

Vícebajtová čísla uložená v souboru jsou vždy v big-endian pořadí. Bezprostředně za hlavičkou se nachází *kořenový uzel*. Každý uzel začíná dvěma bajty obsahujícími délku dat uzlu v bajtech (v kořenovém uzlu našeho příkladu 34). Data uzlu obsahují hodnoty atributů, kódované podle typů atributů deklarovaných v hlavičce. Pro typy TUInt16 a TSInt16 je atribut uložen jako dva bajty obsahující 16bitové celé číslo bez znaménka nebo se znaménkem (dvojkový doplněk). Atributy typu TString jsou ASCII řetězce proměnné délky kódované jako dva bajty určující počet znaků, následované znaky, každý uložený v jednom bajtu. Pro TLink je uloženo 8 bajtů obsahujících pozici jiného uzlu v souboru. Částečný výpis ukazuje, že kořenový uzel nese hodnoty {1000, −1000, "Hello", "World", 0x2C, 0x1234}.

Binární soubor je reprezentován třídou `BinaryFile`, která poskytuje následující metodu, jež načte `len` bajtů do bufferu `buf`, začínajícího na absolutní pozici `filepos` v souboru (znaménkové typy se používají kvůli kompatibilitě s Javou):

```
void readBytes (long filepos, byte[] buf, int len); // Java, C#
void readBytes (std::int64_t filepos, unsigned char* buf, int len); // C++
```

1. Napište deklaraci třídy `Reader`, včetně jejích privátních dat a implementace následujících metod: Konstruktor obdrží otevřený objekt `BinaryFile` jako parametr; okamžitě načte hlavičku. Metoda `attributes` vrací počet atributů. Metoda `getType` vrací typ *i*-tého atributu, číslováno od nuly. Metoda `getRoot` vrací pozici kořenového uzlu. Metoda `readNode` vrátí nový objekt `Node` reprezentující uzel uložený na pozici `filepos`. Objekt je vrácen odkazem v Java/C#, ale hodnotou v C++. Data uzlu se načítají ze souboru pouze touto funkcí.

2. Deklarujte a implementujte privátní datové prvky a konstruktor třídy `Node`. Třída má ukládat data uzlu jako pole bajtů, tj. tak, jak jsou uložena v souboru. Navíc má obsahovat strukturu, která umožní lokalizovat libovolný atribut v konstantním čase. Konverze bajtů na celá čísla nebo řetězce se provádí pouze při volání odpovídajících metod `get...` třídy `Node`.
3. Napište implementaci následujících metod třídy `Node` pro získání hodnoty celočíselných atributů se znaménkem a bez znaménka (16bit), použijte přitom pouze vestavěné operátory zvoleného jazyka:

```
int getUInt16(int i);
int getInt16(int i);
```

Každá metoda čte i -tý atribut; je odpovědností volajícího zajistit, že volání odpovídá typu atributu. Typ `int` se považuje za dostatečně velký, aby obsahoval rozsah $\langle -32768, 65535 \rangle$.

Náčrt řešení.

(0) Dekódování výpisu. Hlavička: 00 06 = 6 atributů, pak typy 48 68 53 53 4C 4C = `TUInt16`, `TSInt16`, `TString`, `TString`, `TLink`, `TLink`. Kořenový uzel začíná na offsetu 8 (za 2 B počtu a 6 B typů), jeho délka je 00 22 = 34 B. Data (big-endian): 03 E8 = 1000; FC 18 = $0xFC18 - 2^{16} = -1000$; délka 5 + Hello; délka 5 + World; 8 B ... 00 2C = $0x2C$; 8 B ... 12 34 = $0x1234$. Spotřeba $2+2+7+7+8+8 = 34$ B sedí.

(1) Reader. Konstruktor přečte počet atributů a typy; `getRoot` vrací offset hned za hlavičkou. `readNode` přečte 2 B délky a pak data uzlu.

```
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));
    }
};
```

(2)–(3) Node. Data drží jako pole bajtů; v konstruktoru předpočítá offsety atributů (kvůli proměnné délce řetězců), takže i -tý atribut najde v $O(1)$. `get` metody skládají big-endian.

```
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; }
};

```

(Náčrt doplněn k výkladu okruhu; otázka nemá v PDF řešení, dekodování i kód ověřeny skriptem `verify_szz.py`.)

10 Semafor mezi procesy

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

Zadání.

Implementujte dvojici funkcí s tímto rozhraním:

```

void send(byte ch);
byte receive();

```

Účelem těchto funkcí je realizace jednosměrného proudu bajtů mezi dvěma procesy. Součástí proudu je FIFO-buffer s kapacitou N bajtů. Funkce `receive`, volaná procesem příjemce, čeká, dokud v bufferu není alespoň jeden platný bajt, poté tento bajt odebere a vrátí jej. Funkce `send`, volaná z procesu odesilatele, čeká, dokud v bufferu není alespoň jedna volná pozice, poté do bufferu vloží parametr `ch`.

Vášim úkolem je implementovat funkce `send` a `receive` pouze pomocí těchto součástí:

- Semaforey (pracující přes hranice procesů)
- Implementace bufferu s kapacitou N , která poskytuje následující rozhraní:

```

void buffer_push(byte ch);
byte buffer_pop();

```

Funkce `buffer_push` (volaná procesem odesilatele) a funkce `buffer_pop` (volaná procesem příjemce) jsou atomické (tj. mohou být bezpečně volány současně) a umožňují předávání bajtů přes hranice procesů (např. pomocí sdílené paměti a atomických instrukcí). Tyto funkce však neposkytují synchronizaci (vždy se vracejí okamžitě). Funkce `buffer_push` nesmí být volána, pokud je buffer plný, a funkce `buffer_pop` nesmí být volána, pokud je buffer prázdný.

Vášim úkolem není řešit vytvoření semaforů a bufferu, předpokládejte, že oba procesy mohou přistupovat k těmto objektům, jako by existovaly uvnitř těchto procesů.

1. Napište implementaci funkcí `send` a `receive` v některém z jazyků C, C++, C# nebo Java. Implementace nesmí používat žádné statické nebo globální proměnné ani žádné další knihovny kromě semaforů a výše popsaného bufferu. Implementace nesmí využívat aktivní čekání (polling). Specifikujte také počáteční stav(y) semaforu/ů (v závislosti na kapacitě bufferu N); buffer je na počátku prázdný.
2. Definujte invariant(y) platné pro stav(y) semaforu/ů ve vztahu k počtu platných elementů v bufferu (a jeho kapacitě N). Všechny tyto invarianty musí být platné po celou dobu mezi voláními funkcí `send` nebo `receive`.
3. Vysvětlete, jak se platnost invariantů mění uvnitř funkcí `send` a `receive` (např. pomocí pozměněných invariantů platných mezi jednotlivými příkazy uvnitř těchto funkcí). Pro jednoduchost předpokládejte, že tyto funkce nejsou volány zároveň.

Náčrt řešení.

```

// capacity_semaphore initialized to N
// content_semaphore initialized to 0
//
// Invariant 1: capacity_semaphore = N - elements_in_buffer
// Invariant 2: content_semaphore = elements_in_buffer

void send(char ch)
{
    capacity_semaphore.down();
    // Inv 1 skewed by +1, Inv 2 valid
    buffer_push(ch);
    // Inv 1 valid, Inv 2 skewed by -1
    content_semaphore.up();
}

char receive()
{
    content_semaphore.down();
    // Inv 1 valid, Inv 2 skewed by +1
    char ch = buffer_pop();
    // Inv 1 skewed by -1, Inv 2 valid
    capacity_semaphore.up();
}

```

Doplňující vysvětlení (nad rámec oficiálního náčrtu): V náčrtu funkce **receive** chybí na konci **return ch;**. Komentáře „skewed by ± 1 “ říkají, o kolik se invariant mezi příkazy rozchází: po **capacity_semaphore.down()** je hodnota semaforu o 1 menší, než by odpovídalo počtu prvků (jedno místo je už rezervované, ale ještě nezaplněné); **buffer_push** invariant 1 srovná a invariant 2 je naopak o 1 pozadu, dokud ho **content_semaphore.up()** nesrovná. Funkce **receive** je symetrická. Další zámek není potřeba, protože **buffer_push** a **buffer_pop** jsou podle zadání atomické a semaforey zaručují, že se push nevolá na plný a pop na prázdný buffer.