

Korespondenční Seminář z Programování

36. ročník

KSP

Duben 2024

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám páté číslo hlavní kategorie 36. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou open-datovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.



Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.



Odměny & na Matfyz bez přijímaček


Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepek na notebook a možná i další překvapení.

Termín série: neděle 23. června 2024 ve 32:00 (tedy další ráno v 8:00)

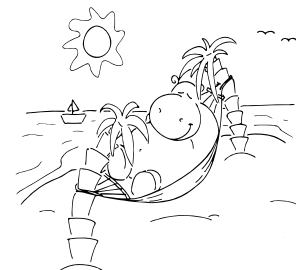
Odevzdávání: Přes web na adrese <https://ksp.mff.cuni.cz/h/odevzdavtko/>

Značky úloh:  Lehčí úloha (či její část) vhodná pro začátečníky  Praktická open-data úloha

 Úloha, u které doporučujeme začít se do kuchařky  Seriálová úloha


 Experimentální (neobvyklá) úloha

Odměna série: Sladkou odměnu si vyslouží ten, kdo získá alespoň 2 body z každého dílu seriálu.



Pátá série třicátého šestého ročníku KSP

36-5-1 Opravování opravy vzducholodi 10 bodů

 Organizátoři KSP se radují. Aby ne, když jim přišlo celkem N řešení úlohy 36-4-1!¹ Bojí se jen, že jim bude trvat dlouho všechna řešení opravit.

Organizátoři se proto pokusili vytrénovat jazykový model, který by řešení opravil za ně. Moc úspěchu ale neměli: nejen že se jim nepodařilo modelu vysvětlit vzorové řešení, ale navíc se bojí, že když některý z účastníků nevyhnutelně vymyslí nějaké originální, netradiční řešení, tak ho jazykový model nepozná a zamítne. Podařila se jim však jedna věc: model naučili zhodnotit složitost a čitelnost daného řešení a na základě toho přesně spočítat, jak dlouho bude organizátorovi trvat ho opravit ručně.

Existuje K organizátorů, kteří mají čas řešení opravovat. Plánují se sejit a řešení si mezi sebou rozdělit. Každý organizátor postupně opraví všechna svá řešení, každé přesně za dobu předpovězenou jazykovým modelem. Opravování bude hotovo, jakmile svá řešení doopraví poslední z organizátorů.

Bohužel došel inkoust ve stroji, který řešení orazítkovává čarovými kódy, takže se organizátoři bojí, že se jim řešení pomíchají. Řešení si proto seřadili abecedně podle jména autora a každému organizátorovi plánují přidělit nějaký souvislý úsek.

Organizátoři by rádi řešení opravili co nejrychleji, aby mohli spolu jít na zmrzlinu. Poradí jim, jak si mají řešení rozdělit?

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku budou dvě přirozená čísla: počet řešení N a počet organizátorů K . Na druhém řádku bude N přirozených čísel T_1 až T_N , časy potřebné k opravě jednotlivých řešení v mikrosekundách. Řešení jsou uvedena v tom pořadí, jak si je organizátoři abecedně seřadili.

Formát výstupu: Za každého opravujícího organizátora vypíšte jeden řádek s dvěma čísly a a b , číslo prvního a posledního řešení (včetně), které má daný organizátor opravit. Těchto řádků můžete vypsát nejvýše K .

Ukázkový vstup:


6 3
4 1 6 10 1 2

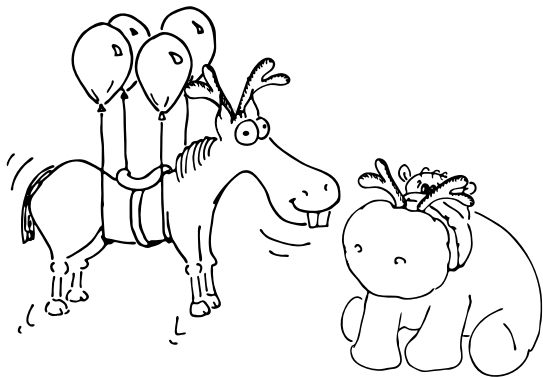
Ukázkový výstup:

1 3
4 4
5 6

Vysvětlení ukázkového vstupu: Prvnímu organizátorovi bude opravování trvat 11 mikrosekund, druhému 10 mikrosekund a třetímu 3 mikrosekundy. Opravování tedy bude hotovo za 11 mikrosekund, což je tak rychle, jak to jen jde.

¹ <http://ksp.mff.cuni.cz/viz/36-4-1>

 Honza se Šimonem byli ze školy vysláni na Konferenci sudokopytných polyglotů v Hrochschwabu, aby tam předali své znalosti hřečtiny. Nejprve ale potřebují vymyslet nějakou rozumnou trasu, kterou se dostat do tak daleké krajiny. Zároveň by si po cestě rádi zopakovali jazyky ostatních zemí. Pokud se totiž účastník konference zvládne domluvit s jakýmkoliv jiným účastníkem v jeho rodné řeči, vyhraje zlaté parohy.



Aby se navzájem motivovali ve cvičení jazyků, domluvili se Honza a Šimon na následujícím postupu: oba nejprve začnou putování v jejich domovském městě Hroška. Jakmile se jejich cesty rozdělí, začnou se oba samostatně učit nějaký nový jazyk. Až se jejich cesty zkříží, promluví si spolu tímto jazykem, obohativše se navzájem díky různým přístupům k učení. Jakmile se po nějaké chvíli zase rozejdou, celý proces se zopakuje s dalšími novými jazyky, a tak dále, dokud se jim nepodaří setkat se v Hrochschwabu.

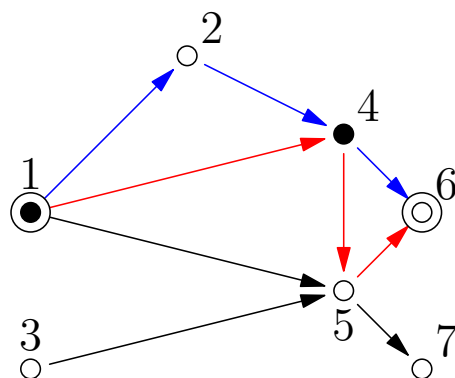
Oba kamarádi by rádi své cesty naplánovali tak, aby se během nich naučili co nejvíce jazyků. Jinými slovy, chtějí se na cestách co nejvícekrát rozejít a zase sejít. Pomůžete jim?

V našem světě existuje celkem N měst očíslovaných 1 až N podle toho, jak vysoko se nachází – nejnižší položené město má tedy číslo 1. Jelikož chodit z kopce je nuda, cesty mohou vést pouze z níže položených měst do vyšších. Z i -tého města potom vedou jednosměrky do n_i dalších měst, a to pouze těch, které mají číslo větší než i . Hroška má číslo v_s a Hrochschwab má číslo v_c , přičemž $v_s \neq v_c$.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu dostanete zadané hodnoty N, v_s, v_c . Na dalších N řádcích obdržíte postupně popisy jednotlivých měst: i -tý řádek začíná číslem n_i . Za ním následuje n_i vzestupně řazených čísel měst, do kterých vede z i -tého města jednosměrka.

Formát výstupu: Nechť vámi nalezená Honzova a Šimonova cesta navštěvuje H , respektive S měst. Na prvním řádku vypíšete H mezerami oddělených čísel měst popisujících postupně Honzovu cestu, tedy prvním číslem musí být v_s a posledním v_c . Na druhém řádku vypíšete ve stejném formátu Šimonovu cestu. Pro tyto dvě cesty musí platit, že se na nich oba kamarádi co nejvícekrát rozejdou a zase potkají. Cesty nemusí být hranově disjunktní, je tedy povoleno, aby občas mezi dvěma městy cestovali Honza a Šimon spolu. Pokud existuje více stejně dobrých cest, vypíšete libovolnou z nich. Máte zaručeno, že mezi Hroškou a Hrochschwabem existuje aspoň jedna cesta.



Ukázkový vstup:

```
7 1 6
3 2 3 4
1 4
1 5
2 5 6
2 6 7
0
0
```

Ukázkový výstup:

```
1 2 4 6
1 4 5 6
```

Vysvětlení ukázkového vstupu: Šimonovy a Honzovy cesty se rozejdou (a někdy později i sejdou) celkem dvakrát: nejprve hned po městě č. 1 a následně po městě č. 4.

36-5-3 Všechny cesty vedou do Říma 12 bodů

Adam a Kačka se rozhodli udělat si pochod z Benátek do Říma. Sbalili si, nasedli na vlak a vyjeli do Benátek. Když jeli kolem hranic, tak se Adam rozhodl si protáhnout nohy a projít se po vlaku. Nicméně chvíli potom rozpojili vlak, a když se Adam snažil domluvit s průvodčím, dostalo se mu pouze odpovědi: „Non riesco a capirti.“

Nyní je Kačka v Benátkách a Adam je v Miláně. Nicméně i přesto se rozhodli udělat pochod do Říma. To udělají tak, že v současném městě zkusí se svou omezenou schopností italštiny zjistit, kterým směrem je Řím. Poté tímto směrem půjdou, dokud nedojdou do dalšího města, kde proces zopakují. Kačku a Adama nyní zajímá, kde se poprvé potkají.

Trochu formálněji: Vrcholy grafu Itálie tvoří města, kde každý vrchol má právě jednu výstupní hranu. Pokud začneme v libovolném vrcholu, a budeme z něj následovat hrany, časem dojdeme do Říma. Kačka stojí ve vrcholu b a Adam ve vrcholu m . Chtěli bychom najít první vrchol, kterým projdou oba.

Nicméně graf Itálie není malý, proto požadujeme, aby algoritmus kromě uložení grafu samotného *používal jen konstantně mnoho paměti*. Navíc pokud se Adam a Kačka budou se svou omezenou schopností italštiny ptát místních na směr do Říma a vždy se tímto směrem vydají, nejspíš nepůjdou úplně přímou cestou.

Proto určujte složitost algoritmu vzhledem k těmto parametrům:

- S_b – Vzdálenost Benátek a prvního společného města
- S_m – Vzdálenost Milána a prvního společného města
- R_b – Vzdálenost Benátek a Říma
- R_m – Vzdálenost Milána a Říma
- N – Počet vrcholů
- M – Počet hran

Například, kdybychom měli následující algoritmus, který je sice funkční, ale **nepoužívá konstantní množství paměti**:

- Najdeme všechny vrcholy na cestě B z Benátek do Říma $\mathcal{O}(R_b)$
- Najdeme všechny vrcholy na cestě M z Milána do Říma $\mathcal{O}(R_m)$
- Pro každý vrchol z B se podíváme, jestli se vyskytuje na cestě M . První, který se vyskytuje je náš hledaný. Protože takto budeme zkoušet vrcholy z B od začátku do prvního společného města, složitost bude $\mathcal{O}(S_b R_m)$

Celková časová složitost tedy bude $\mathcal{O}(R_b + S_b R_m)$.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: <http://ksp.mff.cuni.cz/viz/tinfo>

36-5-4 Hackovací soutěž 11 bodů



Kačka na poslední Smršti² navštívila přednášku o háčkování. To ji velmi nadchlo, koupila si háčky všech možných velikostí a příze všech možných barev a začala každou chvíli svého volného času vyplňovat háčkováním. Kromě řetízkového stehu se naučila háčkovat různé dlouhé sloupky, ubírat a přidávat oka a vyzkoušela si i některé složitější vzory. O pár dní později už byl její pokoj plný malých háčkových chobotniček.

Dokázete si představit Kaččino nadšení, když na chodbě ve škole zahlédla plakát s velkým nápisem „Hackovací soutěž“. Samozřejmě dlouho nemeškala, na uvedené adrese se přihlásila a začala ještě více trénovat, aby ji na soutěži nic nepřekvapilo.

První podezření, že se někde stala chyba, pojala Kačka, když přicházela na místo soutěže. Místo lidí s košíky barevné příze, háčky a různobarevnými kusy oblečení všude okolo sebe viděla ostré hochy, ostrá děvčata a další ostré bytosti v černých mikinách, kteří u sebe neměli ani háčky, ani přízi, ale velké těžké staré notebooky. Další dávku podezření pojala na recepci, kde, když řekla, že přišla na „Háčkovací soutěž“, se na ní dívali poněkud zvláště, než ji navedli správným směrem.

Všechno se vyjasnilo, až když soutěž začala a Kačka si přečetla zadání první úlohy. Vůbec totiž nebyla na *Háčkovací*, ale na *Hackovací* soutěži! Rozhodla se ale, že se jen tak jednoduše nevzdá a vyřeší alespoň jednu úlohu. Pomůžete jí?

Na serveru běží následující program:

```
void print_flag1() { /* ... */ }
void print_flag2() { /* ... */ }
char skutecneheslo[40];
int main() {
    // ...

    bool ok = false;

    char heslo[40];
    printf("Zadejte heslo: ");
    gets(heslo);
    if (strcmp(heslo, skutecne_heslo) == 0)
```

```
        ok = true;
    if (ok)
        print_flag1();
    else
        printf("\nŠpatné heslo.\n");
}
```

Některé části kódu jsou úmyslně vynechané, celý zdrojový kód najdete zde.³

K řešení máte k dispozici přímo zkompileovaný program,⁴ takový, jaký běží na serveru.

Vášim úkolem je zařídit, aby se spustily `print_flag1()` a `print_flag2()`, přičemž máte přístup pouze k standardnímu vstupu a výstupu programu, které jsou připojené přímo na TCP socket na adrese `vm.kam.mff.cuni.cz` a portu 13337. Na ten se můžete připojit například pomocí příkazu `nc vm.kam.mff.cuni.cz 13337` nebo knihovny `socket` v Pythonu.

K řešení se vám bude hodit mít k dispozici nějaké Linuxové prostředí, nějaký *disassembler* (není potřeba nic složitějšího jako IDA nebo Ghidra, bohatě vám postačí `objdump -S`), abyste se mohli podívat, na jakých adresách jsou části programu uložené a *debugger*, například `gdb`, abyste se mohli podívat, co program dělá, když mu dáte různé vstupy. Zajímavé také může být prozkoumat knihovnu `pwn-tools`, ale tuhle úlohu zvládnete vyřešit i bez ní. Nakonec, nezapomeňte si přečíst naši novou kuchařku o práci s pamětí, která vychází součástí této série. Kdybyste narazili během řešení na nějaký problém, nebojte se zeptat na Discordu nebo emailem. Na Discordu ale dávejte pozor, abyste neodhalili část postupu řešení.

Tohle je experimentální open-data úloha. Když úlohu vyřešíte, vypadnou na vás dvě různé *vložky* ve formátu `ksp{.*}`. Každá je správnou odpovědí na jeden ze vstupů v odevzdávacím systému. Odpověď na první vstup vypisuje funkce `print_flag1()` a na druhý vstup funkce `print_flag2()`. Když se vám podaří získat jenom jednu z nich, dostanete jen část bodů.

36-5-X1 Superstring 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezákáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Byl jednou jeden Medvěd a ten měl šuplík, kde schraňoval svou sbírku zajímavých řetězců. Teď zatřídil pár nových úlovků, ale co to? Šuplík už nejde zavřít. Napadlo ho, že když mají řetězce spoustu společných částí, mohl by je rozebrat na kousíčky a složit z nich jediný řetězec, v němž se budou nacházet všechny původní podřetězce. Jak ale takový řetězec najít?

Navrhněte algoritmus, který dostane n řetězců o celkové délce m znaků z nějaké malé konečné abecedy Σ . Na výstupu pak vypíše *nejkratší* možný řetězec, jehož (souvislým) podřetězcem bude každý ze zadaných řetězců.

Jelikož to v polynomiálním čase nikdo neumí (třeba budete první), spokojíme se s časovou složitostí $\mathcal{O}(2^n \cdot \text{poly}(m, |\Sigma|))$.

² <https://ksp.mff.cuni.cz/akce/smrst/>

³ <https://ksp.mff.cuni.cz/h/ulohy/36/36-5-4-vuln.c>

⁴ <https://ksp.mff.cuni.cz/h/ulohy/36/36-5-4-vuln>

Recepty z programátorské kuchyně: Práce s pamětí

V této kuchařce se, poněkud netradičně, nepodíváme pod pokličku nějakého algoritmu, ale povíme si něco o tom, co se děje na pozadí, když se v našem programu rozhodneme vytvořit nějakou proměnnou, číst z ní, nebo do ní zapisovat.

Strojový kód, instrukce a registry

Jak asi víte, procesory počítačů neumí přímo vykonávat jednotlivé řádky našich programů a potřebujeme přivolat na pomoc nějaký kompilátor nebo interpret, který z našeho programu vyrobí tzv. *strojový kód*, což je dlouhý seznam jednoduchých *instrukcí*, kterým už procesor rozumí. Takové instrukce například vezmou čísla ze dvou *registru*, udělají na nich nějakou aritmetickou operaci, a pak výsledek uloží do jiného registru.

Takový *registr* je našim dobře známým proměnným velmi podobný v tom, že se jedná o krabičku uvnitř procesoru, kam si můžeme uložit nějaké jedno číslo a později si ho odtamtud zase přečíst. Bohužel jich ale každý procesor má jen pár – počet velmi závisí na konkrétním procesoru, ale ty, které najdeme ve většině dnešních počítačů, jich mají jen 16 všeobecných (a pak ještě pár specializovaných, ale těmi se s dovolením nebudeme zabývat). Na velmi jednoduché programy nám to možná bude stačit, ale hned jak budeme chtít pracovat se seznamem čísel delším než 16, už budeme mít problém. Registry mají také omezenou velikost, obvykle 64 nebo 32 bitů, což odpovídá tomu, že se procesorům říká „64-bitové“ nebo „32-bitové“.

Paměť

Proto mají kromě registrů procesory také k dispozici paměť. Tu si můžeme představit jako dlouhatánský seznam očíslovaných políček, do kterých se dají opět, jak jinak, ukládat čísla. Číslům, kterými jsou označena políčka, se obvykle říká *adresy*, číslům uloženým v políčkách *hodnoty* a často mluvíme o *hodnotě paměti na adrese X* nebo také *obsahu paměti na adrese X*. Všechny *hodnoty* jsou ve skutečnosti jen posloupnosti bytů. Všechna data, která chceme ukládat, včetně celých čísel, floatů, obrázků nebo i instrukcí procesoru, tak musíme do jednotlivých bytů nějak zakódovat. Adresy mají velikost jednoho *slova*, která odpovídá velikosti registrů. Každá⁵ adresa odpovídá jednomu bytu paměti, ale⁶ objekty uložené v paměti mohou být různě velké. Adresa pak ukazuje na jejich první byte. Adresy bývají často zarovnané na slova, ale není to povinné. Jediné, co taková paměť umí, je na požádání vrátit hodnotu, která se nachází na dané adrese, nebo na danou adresu konkrétní hodnotu zapsat.

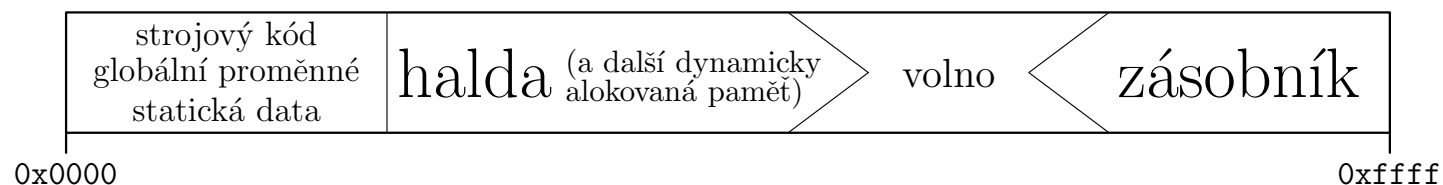
Procesor nám proto nabízí instrukce, které nám umožňují načíst hodnotu z pevné (tzv. *okamžité*, angl. *immediate*) adresy a uložit ji do nějakého registru. Taková adresa je pak pevně zabudovaná ve strojovém kódu a pokaždé, když program spustíme, bude procesor pracovat s tou stejnou adresou. To na některé věci stačí, ale složitější věci s tím nevyřešíme. Nejjednodušším příkladem, kdy pevné adresy nestačí, je třeba práce s polem – museli bychom do strojového kódu zapéct adresu každé buňky, navíc bychom nemohli pracovat například s poli proměnlivé velikosti. Proto existují také instrukce, které si přečtou adresu, ze které mají hodnoty načítat, z nějakého dalšího registru. Dokonce máme i instrukce, které dělají: „Načti hodnotu z registru X, přičti k ní tuhle pevnou hodnotu a z výsledné adresy načti hodnotu do registru Y“. To se stane třeba ve chvíli, kdy v Céčku přistoupíte k nějakému políčku ve struktuře – víme adresu celé struktury a zajímá nás konkrétní políčko v ní.

Části paměti

Teď už nám zbývá jen vyřešit, jak se mají jednotlivé programy a jejich části domluvit, na jaké adresy mají uložit obsah svých proměnných, aby si vzájemně nepřekážely. K tomu slouží tři různá místa: tzv. *datová paměť*, kde se ukládají globální proměnné, a poté *halda* a *zásobník*. Pozor, ty nemají nic společného s datovými strukturami *halda* a *zásobník*, a byť se občas mohou chovat podobně, jsou to odlišné koncepty.

Do datové paměti, se kromě globálních proměnných ukládá také například strojový kód programu. Důležité je, že ve chvíli, kdy generujeme strojový kód, už víme, kolik celkem budeme mít globálních proměnných, takže všemu, co se nachází v datové paměti, můžeme přiřadit pevné adresy a ty si pevně zapsat do kódu. Virtualizace paměti nám pak zařídí, aby dva různé programy mohly používat stejné pevné adresy pro datovou paměť a nesrazily se (jak virtualizace funguje, si ukážeme později).

Na lokální proměnné nám proto zbývá právě halda se zásobníkem. Vzhledem k tomu, že předem nevíme, kolik paměti budeme potřebovat, je nutné, abychom uměli haldu i zásobník postupně nafukovat, jak v nich budou přibývat data. Proto haldu dáme na začátek volné paměti a zásobník na samotný konec, přičemž haldu pak můžeme rozšiřovat směrem k vyšším adresám a zásobník směrem k nižším. Často se říká, že „zásobník roste dolů“, čímž popisujeme přesně tuto skutečnost – čím více hodnot na zásobníku bude, tím menší bude jejich adresa.



⁵ Z historických důvodů se můžete také (obzvlášť v podkladech od společnosti Intel) setkat s tím, že se termín *slovo* nebo také *WORD* označuje velikost přesně 16-bitů, *double WORD* nebo *DWORD* pak značí přesně 32 bitů a *quad WORD* nebo *QWORD* značí 64 bitů. My se však budeme držet toho, že velikost *slova* vždy odpovídá velikosti registru daného procesoru.

⁶ To je mimochodem důvod, proč 32-bitové počítače nemohou mít více než 4 GB paměti. Paměť je adresována celkem 2^{32} adresami, má tedy celkem 2^{32} bytů, což je právě téměř 4 GB.

Volání funkcí a zásobník

Než se dostaneme k tomu, jak funguje zásobník, povíme si něco o tom, jak se ve strojovém kódu volají funkce. Strojový kód máme uložený v datové paměti (podobně jako globální proměnné, jak jsme si popsali výše). To znamená, že už v době kompilace známe adresu všech instrukcí, speciálně tedy i adresu začátku všech funkcí. Když voláme funkci, stačí nám tedy „skočit“ na předem známé místo v datové paměti a začít vykonávat kód odsud (instrukci provádějící tento skok se většinou říká `jump`). Ale kam pak skočit, když funkce skončí? Bylo by pěkné si ve chvíli, kdy voláme nějakou funkci, poznamenat, z jakého místa ve strojovém kódu jsme přišli, abychom se tam pak po skončení volané funkce mohli vrátit. Nejjednodušší by bylo si na to vyhradit jeden celý registr, tam bychom ale narazili na problém, kdybychom potřebovali uvnitř volané funkce volat nějakou jinou funkci. Chtěli bychom si pořídit datovou strukturu, které můžeme při každém volání funkce předhodit aktuální adresu, a při návratu z funkce z ní vytáhnout tu adresu, kterou jsme tam přidali jako poslední. A někteří z vás už si bezesporu všimli, že to je přesný popis chování zásobníku.

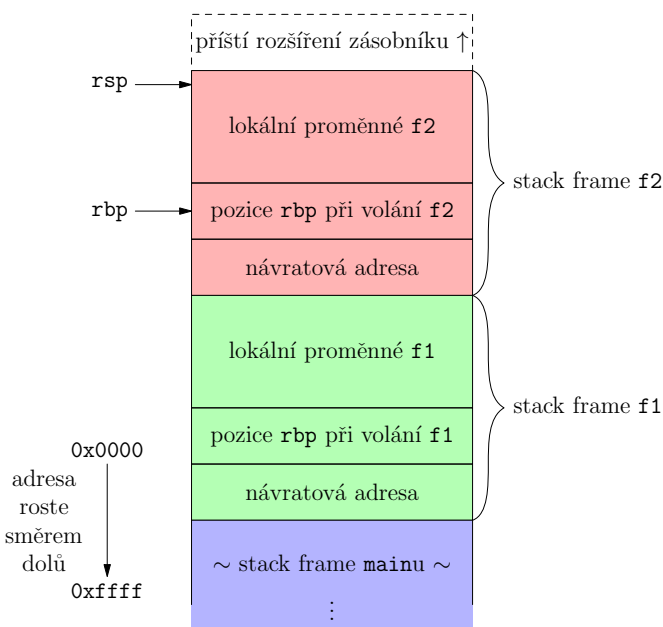
Přeci jen má *zásobník* něco společného se stejnojmennou datovou strukturou. A jak ho implementujeme jen pomocí pár jednoduchých instrukcí? Prostě mu vyhradíme nějaké předem dané místo v paměti (v tom nejjednodušším případě, jak jsme se dočetli výše, na úplném konci) a vyhradíme si jeden registr (v terminologii `x86mu`⁷ obvykle říkáme `rsp` – Register for Stack Pointer), kde si budeme ukládat ukazatel (adresu) na místo v paměti, kde je aktuálně vršek zásobníku. Když chceme na zásobník přidat hodnotu *X*, prostě `rsp` zmenšíme o jedničku (připomeňte si, že zásobník roste dolů) a na adresu, kam `rsp` ukazuje, zapíšeme *X*. Při čtení postupujeme přesně obráceně.

Když už si na zásobník ukládáme *návratové adresy* (*return address*), mohli bychom tam začít dávat i lokální proměnné, jen si musíme dát na pár věcí pozor. Vytváření nových lokálních proměnných nebude složité, můžeme je přidávat na zásobník beze změny. Jak ale později zjistíme, kde máme proměnnou uloženou? Zapéct její adresu do kódu nemůžeme, neboť může být pokaždé někde jinde. Můžeme si pamatovat, že jsme ji uložili na adresu, kam ukazuje `rsp`, ale tak získáme jen adresu úplně poslední lokální proměnné. Řešením je pamatovat si, jak daleko od konce zásobníku naše proměnná je – tomu se říká *offset* vůči `rsp`. Pokud bychom si ale uprostřed funkce poříдили novou lokální proměnnou, posuneme tím `rsp`, a tím pádem i offset všech ostatních lokálních proměnných na zásobníku. Je také obvyklé, že se všechny lokální proměnné vytvoří na začátku funkce najednou a po dobu běhu funkce už se pozice `rsp` nemění.

Stejně tak, jak se dostaneme k návratové adrese, když jsme si teď zaházeli volací zásobník lokálními proměnnými? Úplně stejně, jako k lokálním proměnným, které nejsou na vrcholu zásobníku. Prostě si při generování kódu pamatujeme, kolik aktuálně máme na zásobníku lokálních proměnných, a víme, že hned pod nimi je návratová adresa. Těsně předtím, než skočíme z funkce ven, pak musíme taky zásobník uklidit, a to tak, že `rsp` posuneme na místo, kde byl před voláním funkce, tedy hned pod návratovou adresu.

Tohle dopočítávání během generování strojového kódu je sice hezké a stačí k funkčnímu modelu, ale je to nepřehledné a jednou za čas se hodí mít možnost si i strojový kód přečíst, což se dělá špatně, když všechno odpočítáváme od `rsp`. Proto se téměř vždy používá i druhý registr (kterému se v `x86` říká `rbp` – Register for stack Base Pointer), který vždy ukazuje na začátek *stack frame*, hned pod ním je návratová adresa. *Zásobníkový rámec*, nebo častěji *stack frame*, je ta část paměti, která obsahuje všechna data pro jedno konkrétní volání funkce. Při volání funkce je posun `rbp` jednoduchý, prostě jeho hodnotu nastavíme na aktuální hodnotu `rsp`, protože právě tam budeme zapisovat návratovou adresu. Při návratu z funkce je to horší, musíme totiž zjistit, co bylo v `rbp` předtím, než jsme ho změnili. Takový problém jsme ale už přeci řešili, prostě při volání předchozí hodnotu `rbp` vložíme na zásobník za návratovou adresu a tam si ji pak také při návratu přečteme.

Na obrázku níže vidíte, jak bude vypadat zásobník, pokud ve funkci `main` zavoláme funkci `f1` a uvnitř té zavoláme funkci `f2`. Šipky ukazují aktuální pozice `rsp` a `rbp` ve chvíli, kdy se nacházíme uvnitř `f2`.



Mimochodem, většina dnešních procesorů poskytuje na volání funkcí samostatné instrukce, v `x86` je to konkrétně `call`, která přidá na zásobník adresu následující instrukce (té, na kterou se pak budeme vracet) a skočí na pevně danou adresu funkce. Instrukce `ret` poté z vrcholu zásobníku jednu adresu sundá a skočí na ni. Stejně tak pro práci s `rbp` poskytuje instrukční sada `x86` instrukce `enter` a `leave`.

Halda a dynamická alokace

Teď už nám zbývá jen povědět, jak funguje halda. Ta už má s datovou strukturou stejného jména společného mnohem méně. Je to místo, kde si musí každý svůj prostor v paměti ručně zarezervovat (*naalokovat*) a pak, když už ho nepotřebuje, ho zase uvolnit ostatním (*dealokovat*). Na pozadí je pak nějaký *alokátor*, obvykle součástí standardní knihovny jazyka, který si drží přehled o zaplněnosti haldy (obvykle pomocí *metadat* uložených před nebo za alokovanou částí

⁷ `x86` je název sady instrukcí, která byla poprvé použita pro procesor Intel 8086, od kterého má i název. Od té doby se rozšířila na další procesory a prakticky všechny procesory v počítačích a serverech jsou dnes postavené na jejím 64-bitovém rozšíření `x86-64`. I když autorem `x86` je Intel (a `x86-64` zase AMD), instrukční sadu používají, s malými rozdíly, všichni. Hlavní konkurencí `x86` je skupina instrukčních sad `ARM`, která se používá spíše v mobilních zařízeních, ale některé společnosti experimentují i s použitím na desktopu, například architektura M1 a M2 od společnosti Apple.

paměti) a vyřizuje požadavky na alokaci nebo dealokaci. Dealokovaná paměť se samozřejmě dále používá, takže pokud paměť nevynulujeme, můžeme tam najít data, která tam byla předtím. Nevýhoda haldy je samozřejmě to, že manuální alokace je pro programátora pracná, a pokud někde zapomeneme dealokovat, paměť nám postupně dojde a nejspíše brzy poté spadne celý počítač.

V C se paměť alokuje pomocí funkce `void *malloc(size_t size)`, kde parametr `size` určuje velikost úseku, který chceme naalokovat. Funkce vrátí ukazatel na začátek nově vyhrazeného úseku (což je ve skutečnosti jen adresa), případně `NULL`, pokud alokace selhala. Funkce `void free(void *ptr)` pak umožňuje uvolnit daný úsek k dalšímu použití. Předává se jí ukazatel na začátek úseku, tedy ukazatel, který vrátil `malloc`.

Také je dobré vědět, že halda může trpět *fragmentací* – když alokujeme a dealokujeme různě velké kousky paměti, postupně na haldě vznikají díry, které se nedají znovu naplnit, pokud alokujeme větší úseky, než jsou díry. Tak se nám může stát, že máme na haldě spoustu volného místa, ale stejně nemáme novou alokaci kam dát.

Virtualizace paměti

Nakonec by se ještě slušelo zmínit, co že je to ta virtualizace paměti, která byla zmíněna výše. Ve zkratce se jedná o mechanismus, pomocí kterého se operační systém postará, aby to z pohledu všech programů vypadalo, že jsou samy na světě – na začátku je z jejich pohledu paměť prázdná a obrovská, neboť každá adresa délky slova je validní, i když je daleko za kapacitou fyzické paměti, kterou má počítač k dispozici. Operační systém si udržuje tzv. *stránkovací tabul-*

ku, kde pro každý kus paměti, který se program rozhodne použít, udržuje místo, kde se tenhle kus paměti vyskytuje v paměti fyzické, a toto místo ve fyzické paměti rozděluje tak, aby každý program měl své. Díky tomu mohou všechny programy zasahovat do paměti, jak chtějí, a nemusí řešit, jestli nezasahují do cizích dat. Při startu každého programu se do této virtuální paměti připraví, mimo jiné, také místo pro haldu a zásobník, takže je běžící program může rovnou začít využívat. Program také nemůže jen tak přistupovat na jakékoliv místo v paměti, ale musí požádat OS, aby mu vyrobil pro dané místo v tabulce stránku ve stránkovací tabulce. To je další věc, kterou na pozadí dělá `malloc`. Kromě alokace samotné si také vyžaduje přidělení nových stránek pokud v těch, které už má přidělené, nenajde místo. Když se stane, že paměť dojde celému počítači a OS už nemá žádnou volnou stránku, kterou by přidělil, vrátí `malloc` `NULL`. Pokud se pokusíme přečíst z nějaké adresy, kterou nám OS nepřidělil, program „vyhodí segfault“. Paměti, kterou nám ale už jednou `malloc` vrátil, můžeme věřit.

To bude z dnešní kuchařky vše. Ukázali jsme si, že převod našich programů na instrukce, kterým procesory rozumí, není vždy tak jednoduchý, jak se může na první pohled zdát. Také jsme si řekli, co jsou to registry, jak funguje paměť a jak si pomocí nich pořídit docela příjemně použitelná primitiva pro proměnné. Dozvěděli jsme se, co je to halda a zásobník v kontextu paměti programu a krátce jsme nastínili, jak funguje virtualizovaná paměť.

Guláš v paměti vám uvařil

Honza Černohorský