Optimalizace nákladu

Termín odevzdání:	**05.04.2021 23:59:59**
Hodnocení:	**30.0000**
Max. hodnocení:	**30.0000** (bez bonusů)
Odevzdaná řešení:	9 / 75
Nápovědy:	0 / 0

Úkolem je realizovat sadu tříd, která budou optimalizovat zisk za přepravu zboží na lodích.

Předpokládáme, že máme k dispozici lodě schopné přepravovat zboží. Každá loď má definovanou maximální nosnost (maxWeight) a maximální objem přepravovaného zboží (maxVolume). Ani jeden z těchto limitů nelze při nakládání překročit. Loď má jednoznačně určený cíl (destination), jedná se o textový řetězec.

V úloze dále vystupují zákazníci. Ti na základě nabídky cíle definují seznam zboží, které má být do zadaného cíle přepraveno. Seznam zboží může mít libovolnou délku (0, 1, 2, ..., N položek), u každého zboží na seznamu je zadaná jeho hmotnost, objem a nabízená cena za přepravu.

Implementovaná třída `CCargoPlanner` bude poptávat registrované zákazníky, vyžádá si od nich seznam zboží k přepravě a určí, jak naplnit lodě zbožím tak, aby byl zisk za přepravu co nejvyšší. Při výpočtu tedy vybere podmnožinu zboží ze seznamů, které dodali zákazníci. Při výběru lze zboží (jednu položku na seznamu) buď naložit celou, nebo vůbec. Například pokud je poptávána přeprava zboží s hmotností 100, objemem 50 a cenou 1000, pak lze buď celé toto zboží naložit (a vyhradit patřičné kapacity lodi) nebo jej nenakládat vůbec. Zboží nelze dále dělit (například naložit hmotnost 50, objem 25 za cenu 500). Protože se jedná o výpočetně náročnou úlohu, bude implementace používat vlákna pro urychlení výpočtů a pro správnou komunikaci se zákazníky.

Vytvořená instance `CCargoPlanner` dostane předepsaný počet vláken pro výpočty (`workThreads`) a počet vláken pro obchodníky (`salesThreads`). Dále bude reagovat na asynchronní volání, která ji informují o lodích, které je potřeba naložit. Instance `CCargoPlanner` bude používána takto:

- vytvoří se instance `CCargoPlanner`,
- zaregistrují se zákazníci (voláním metody `Customer`),
- spustí se vlastní výpočet (voláním metody `Start`). Metoda `Start` dostane v parametrech počet vláken pro obchodníky (`salesThreads`) a počet výpočetních vláken (`workThreads`). Tato vlákna vytvoří a nechá je čekat na příchozí požadavky. Po vytvoření vláken se `Start` vrátí do volajícího,
- testovací prostředí bude opakovaným voláním metody `Ship` zadávat lodě, které je potřeba naložit. Metoda `Ship` může být volaná z hlavního vlákna, případně i z dalších jiných vláken testovacího prostředí. Předpokládá se, že zavolání metody `Ship` je velmi rychlé, volaný si pouze převezme informaci o lodi a zajistí její naložení pomocí vláken obchodníků a pracovních vláken,
- po zadání poslední lodě testovací prostředí zavolá metodu `Stop`. Tímto voláním dává najevo, že již nebude zadávat další lodě. `CCargoPlanner` zajistí naplnění dosud předaných lodí a ukončení všech vytvořených vláken (vlákna obchodníků i pracovní vlákna). Po doběhnutí všech těchto vláken se `Stop` vrátí do volaného,
- je uvolněna instance `CCargoPlanner`.

Použité třídy a jejich rozhraní:

- `CCargo` je třída reprezentující jeden kus přepravovaného zboží. Jedná se o velmi jednoduchou třídu, která zapouzdřuje hmotnost, objem a poplatek za přepravu pro toto zboží.
- `CShip` je třída reprezentující společné rozhraní lodí k naložení (nakládané lodě budou potomci této základní třídy). Loď ukládá informaci o své kapacitě (`m_MaxWeight` a `m_MaxVolume`) a cíli své cesty (`m_Destination`). Tyto atributy jsou vyplněné v konstruktoru. Dále jsou k dispozici metody:
  - `Destination` pro zjištění cíle cesty. Můžete se spolehnout, že každá nakládaná loď bude mít jiný cíl své cesty,
  - `MaxWeight` pro zjištění maximální nosnosti lodě,
  - `MaxVolume` pro zjištění maximálního objemu lodě,
  - `Load` pro naložení vypočteného nákladu. Tuto metodu bude volat výpočetní vlákno, které nalezne optimální zaplnění lodě. Parametrem volání je seznam nákladu, který má být naložen (`vector<CCargo>`). Pořadí nákladu ve vektoru nehraje roli. Pokud existuje více stejně dobrých řešení, lze zvolit libovolné z nich.
- `CCustomer` je třída reprezentující společné rozhraní zákazníků (skuteční zákazníci budou potomci této základní třídy). V rozhraní zákazníka je důležitá metoda `Quote (dest, cargo)` , která zjistí požadavky na přepravované zboží do daného cíle. Tuto metodu budou volat vlákna obchodníků poté, co se objeví nová loď k naložení. Volání této metody může trvat docela dlouho (zákazník se dlouho rozhoduje, co a za kolik chce dopravovat). Metodu `Quote` jednoho zákazníka může najednou volat více vláken obchodníků, například pro různé nakládané lodě.
- `CCargoPlanner` je Vámi implementovaná třída, která bude zapouzdřovat požadovanou funkcionalitu zaplňování lodí. Její veřejné rozhraní musí splňovat:
  - konstruktor bez parametrů inicializuje novou instanci třídy. Zatím nevytváří žádná vlákna,
  - metodu `Customer (x)`, tato metoda zaregistruje zákazníka x,
  - metodu `Start ( salesThr, workThr )`, tato metoda vytvoří zadané počty vláken pro obchodníky a výpočty. Vlákna po vytvoření budou čekat na zadání lodě (lodí) pro odbavení. Volání metody `Start` se ihned po inicializaci vrátí do volajícího,
  - metodu `Ship (x)`, metoda předá novou instanci lodě x k nakládce. Volání metody zařídí předání lodě vláknům obchodníků a výpočetním vláknům, ale uvnitř metody samotné se nebudou provádět žádné časově náročné výpočty. Volání `Ship` se tedy prakticky okamžitě vrátí do volajícího,
  - metodu `Stop`, která počká na odbavení předaných lodí, dokončení všech výpočtů a ukončení vytvořených vláken. Po tomto se volání `Stop` vrátí zpět do volajícího,
  - třídní metodu `SeqSolver(cargo, maxWeight, maxVolume, load)`. Metoda slouží k otestování vlastního algoritmu nakládání lodě. Parametrem volání je seznam zboží `cargo`, z tohoto seznamu bude náklad vybírán, `maxWeight` a `maxVolume` udávající kapacitu lodě a `load` - výstupní parametr, do kterého bude umístěno zboží k naložení (zboží v tomto seznamu nesmí překročit kapacity lodě a musí dávat nejlepší součet poplatků). Návratovou hodnotou metody je suma poplatků za vybraný náklad.
- funkce `ProgtestSolver (cargo, maxWeight, maxVolume, load)` je hotová implementace algoritmu pro výběr zboží k naložení. Funkce má rozhraní stejné jako třídní metoda `CCargoPlanner::SeqSolver`. Vaše implementace může tuto funkci použít pro vlastní výpočet, případně tuto funkci nemusíte používat a výpočet si implementovat sami. V posledním (bonusovém) testu je implementace této funkce v testovacím prostředí úmyslně nefunkční (vrací prázdné seznamy).

Implementace předpokládá vytvoření vláken pro obchodníky a pracovních vláken. Počty těchto vláken jsou zadané ve volání `Start`. Po zadání lodě k naložení (metoda `Ship`) se aktivují vlákna obchodníků. Ze zadané lodi je potřeba získat cíl a poptat všechny registrované zákazníky na jejich seznamy zboží, které chtějí zaslat do tohoto cíle. Vlákna obchodníků proto budou volat metodu `CCustomer::Quote`. Zákazníci mohou na výzvu reagovat rychle (volání `Quote` se okamžitě vrátí), ale jejich odpověď může trvat i velmi dlouho (volání `Quote` je blokující). Je proto třeba zapojit všechna dostupná vlákna obchodníků k obsloužení všech zákazníků / cílů. Zákazník A se na dotazu pro destinaci X může zaseknout na dlouhou dobu (efektivně tedy blokuje příslušné vlákno obchodníka). Ale ten samý zákazník může být paralelně dotazován jiným vláknem obchodníka na destinaci Y, kterou zodpoví okamžitě. V implementaci tedy nelze fixně přidělit obchodníkům jejich lodě / zákazníky, vždy je potřeba práci rozdělovat podle aktuální dostupnosti. Jeden z testů je navržen tak, aby fixní přidělování vedlo k deadlocku.

Až pro danou loď sesbíráte požadavky od všech zákazníků, je potřeba z takto získaného seznamu zboží vybrat náklad, který bude na loď fakticky umístěn. Toto je práce pro pracovní vlákna, kterým problém předají vlákna obchodníků. Vlákna obchodníků nebudou provádět vlastní výpočet, nejsou k tomu určena a mají vlastní povinnosti obsluhovat zákazníky. Pracovní vlákna provedou vlastní výpočet (buď pomocí dodané funkce `ProgtestSolver` nebo si výpočet implementujete svůj) a vybraný náklad umístí na loď (`CShip::Load`). Metodu `CShip::Load` musíte pro každou loď zavolat právě jednou.

Pokud se rozhodnete pro vlastní implementaci, zohledněte následující:

- problém se podobá problému balení batohu. Je navíc zkomplikovaný tím, že musí zohlednit dvě kritéria (hmotnost a objem),
- problém batohu má rekurzivní řešení, které má exponenciální složitost vzhledem k počtu zadaných předmětů. V úloze se zadává 1-128 předmětů (zboží), tedy rekurzivní řešení není časově zvládnutelné,
- pro řešení se dá použít dynamické programování. Kapacity lodí jsou rozumně velké (nosnost i objem jsou čísla řádu tisíců), tedy požadavky na paměť při DP jsou zvládnutelné,
- i tak je ale potřeba dobrý návrh reprezentace dat, aby výpočet byl rychlý a případně aby šel paralelizovat (pro poslední bonusový test). Referenční řešení reprezentuje data pomocí bitových polí, aby se řešení vešlo do dostupné paměti (cca 1 GiB). Využívá mj. toho, že zboží se vybírá ze seznam s délkou nejvýše 128 položek.

Odevzdávejte zdrojový kód s implementací požadované třídy `CCargoPlanner` s požadovanými metodami. Můžete samozřejmě přidat i další podpůrné třídy a funkce. Do Vaší implementace nevkládejte funkci `main` ani direktivy pro vkládání hlavičkových souborů. Funkci `main` a hlavičkové soubory lze ponechat pouze v případě, že jsou zabalené v bloku podmíněného překladu.

Využijte přiložený ukázkový soubor. Celá implementace patří do souboru `solution.cpp`, dodaný soubor je pouze mustr. Pokud zachováte bloky podmíněného překladu, můžete soubor `solution.cpp` odevzdávat jako řešení úlohy.

Při řešení lze využít pthread nebo C++11 API pro práci s vlákny (viz vložené hlavičkové soubory). Dostupný kompilátor g++ verze 8.3, tato verze kompilátoru zvládá většinu C++11, C++14 a C++17 konstrukcí.

Doporučení:

- Začněte se rovnou zabývat vlákny a synchronizací, nemusíte se zabývat vlastními algoritmy řešení zadaných problémů. Využijte dodané řešení - funkci `ProgtestSolver`. Až budete mít hotovou synchronizaci, můžete začít s implementací vlastního algoritmického řešení.
- Abyste zapojili co nejvíce jader, obsluhujte co nejvíce lodí a zákazníků najednou. Je potřeba zároveň přijímat nové lodě, poptávat zákazníky, řešit zaplnění a vyplňovat nalezená řešení. Nepokoušejte se tyto činnosti nafázovat (například nejdříve pouze počkat na všechny lodě, pak začít poptávat zákazníky, ...), takový postup nebude fungovat. Testovací prostředí je nastaveno tak, aby takové "sekvenční" řešení vedlo k uváznutí (deadlock).
- Instance `CCargoPlanner` je vytvářená opakovaně, pro různé vstupy. Nespoléhejte se na inicializaci globálních proměnných - při druhém a dalším zavolání budou mít globální proměnné hodnotu jinou. Je rozumné případné globální proměnné vždy inicializovat v konstruktoru nebo na začátku metody `Start`. Ještě lepší je nepoužívat globální proměnné vůbec.
- Nepoužívejte mutexy a podmíněné proměnné inicializované pomocí `PTHREAD_MUTEX_INITIALIZER`, důvod je stejný jako v minulém odstavci. Použijte raději `pthread_mutex_init()` nebo C++11 API.
- Instance zákazníků a lodí alokovalo testovací prostředí při vytváření příslušných smart pointerů. K uvolnění dojde automaticky po zrušení všech odkazů. Uvolnění těchto instancí tedy není Vaší starostí, stačí zapomenout všechny takto předané smart pointery. Váš program je ale zodpovědný za uvolnění všech ostatních prostředků, které si sám alokoval.
- Metoda zákazníka (`CCustomer::Quote`) je reentrantní a thread-safe. Neobalujte tato volání mutexy, je to zbytečné a zbytečně si tím serializujete své řešení.
- Neukončujte metodu `Stop` pomocí `exit`, `pthread_exit` a podobných funkcí. Pokud se funkce `Stop` nevrátí do volajícího, bude Vaše implementace vyhodnocena jako nesprávná.
- Využijte přiložená vzorová data. V archivu jednak naleznete ukázku volání rozhraní a dále několik testovacích vstupů a odpovídajících výsledků.
- V testovacím prostředí je k dispozici STL. Myslete ale na to, že ten samý STL kontejner nelze najednou zpřístupnit z více vláken. Více si o omezeních přečtěte např. na **C++ reference - thread safety.**
- Testovací prostředí je omezené velikostí paměti. Není uplatňován žádný explicitní limit, ale VM, ve které testy běží, je omezena 4 GiB celkové dostupné RAM. Úloha může být dost paměťově náročná, zejména pokud se rozhodnete pro jemné členění úlohy na jednotlivá vlákna. Pokud se rozhodnete pro takové jemné rozčlenění úlohy, možná budete muset přidat synchronizaci běhu vláken tak, aby celková potřebná paměť v žádný okamžik nepřesáhla rozumný limit. Pro běh máte garantováno, že Váš program má k dispozici nejméně 1 GiB pro Vaše data (data segment + stack + heap). Pro zvídavé - zbytek do 4GiB je zabraný běžícím OS, dalšími procesy, zásobníky Vašich vláken a nějakou rezervou.
- Pokud se rozhodnete pro všechny bonusy, je potřeba velmi pečlivě nastavovat granularitu řešeného problému. Pokud řešený problém rozdělíte na příliš mnoho drobných podproblémů, začne se příliš mnoho uplatňovat režie. Dále, pokud máte najednou rozpracováno příliš mnoho problémů (a každý je rozdělen na velké množství podproblémů), začne se výpočet dále zpomalovat (mj. se začnou hůře využívat cache CPU). Aby se tomu zabránilo, řídí referenční řešení počet najednou rozpracovaných úloh (navíc dynamicky podle velikosti rozpracované úlohy).
- Výpočetně náročné operace musí provádět pracovní vlákna. Počet pracovních vláken je určen parametrem metody `Start`. Testovací prostředí kontroluje, zda Vaše implementace neprovádí výpočetně náročné operace v ostatních vláknech. Pokud to zjistí, Vaše řešení bude odmítnuto.
- Explicitní nastavení počtu pracovních vláken má dobré praktické důvody. Volbou rozumného počtu pracovních vláken můžeme systém zatížit dle naší volby (tedy například podle počtu jader, která můžeme úloze přidělit). Pokud by časově náročné výpočty probíhaly i v jiných vláknech (obsluha zákazníků, příjem asynchronně předaných lodí), pak by bylo možné systém snadno zahltit.