Ocelové plechy

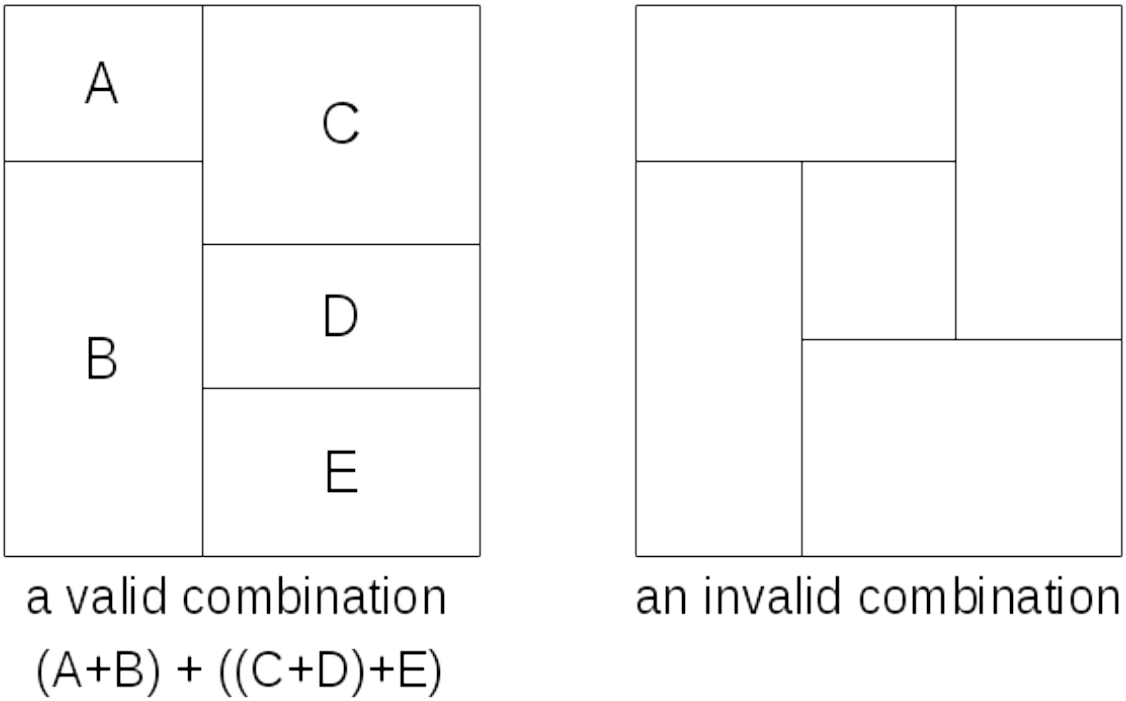
Úkolem je realizovat třídu, která bude umožňovat rychle naceňovat poptávky zákazníků.

Předpokládáme společnost, která se zabývá dodávkou svařovaných ocelových plechů. Tato společnost dostává od svých zákazníků poptávky na výrobu plechů o zadané velikosti a zadané kvalitě. Úkolem je takové poptávky naceňovat a vyplněné je odevzdávat zpět zákazníkům.

Zákazníci vždy odebírají plechy ve tvaru obdélníku (čtverce), jeho velikost je zadaná jako dvojice celých čísel: šířka a délka. Dalším parametrem je typ materiálu (zahrnující jeho tloušťku, pevnost, ...), pro účely našeho zpracování bude materiál identifikovaný celým číslem - identifikátorem materiálu `materialID`. Protože poptávaná velikost plechu nemusí být přímo vyráběna, lze požadované velikosti dosáhnou svařením několika menších plechových tabulí. Svařování může mít různou pevnost, od toho se odvíjí cena sváru. Pro jednoduchost budeme předpokládat, že cena sváru je přímo daná jeho kvalitou. Zákazník tedy v poptávce vyplní požadovanou kvalitu sváru, pro zhotovení toto číslo bude zároveň cenou za jednotku délky sváru. Při výrobě lze menší plechové tabule svařovat, nelze je ale dělit (naše firma nemá k dispozici odpovídající nástroj na řezání plechu, protože na něj nebyla schválena dotace).

Pro výrobu svařených plechů odebírá naše firma prefabrikované výrobky od různých dodavatelů. Každý z dodavatelů dodává výrobky v různých velikostech a za různou cenu. Tedy před vlastním naceněním požadovaného výrobku pro našeho zákazníka firma osloví své dodavatele a vyžádá si u nich ceník pro zadaný materiál (`materialID`). Ceník obsahuje seznam vyráběných velikostí a cenu každého z nich. Ten samý materiál mohou dodávat různí dodavatelé, každý z dodavatelů může mít jiný výrobní program a dodavatelé mohou mít různé ceny. Protože se ale jedná o jakostně stejný materiál, při naší výrobě můžeme libovolně kombinovat prefabrikáty od různých dodavatelů a tím dosáhnout nižší výsledné ceny.

S takto zkombinovanými ceníky a při známé ceně svařování lze vypočítat, za jakou cenu jsme schopni našim zákazníkům dodat poptávané výrobky. Obecný algoritmus výpočtu je dost technický, náš příklad ale předpokládá zjednodušení dané technologií svařování. Dokážeme svařovat pouze dvojice plechů, které mají stejnou délku hrany (viz obrázek), svár musí být vždy přes celou délku hrany. Pro toto zjednodušení lze řešení nalézt v kubickém čase. Protože kubické algoritmy jsou pro větší vstupy zpravidla pomalé, využijeme ke zrychlení výpočtu vlákna.



Vaším primárním úkolem je navrhnout a implementovat třídy, které umožní efektivní fungování vícevláknového řešení problému. Vlastní algoritmus výpočtu nemusíte implementovat, testovací prostředí zpřístupňuje funkci `ProgtestSolver`, kterou lze zavolat a pro výpočet použít. Správné řešení postavené na této dodané funkci projde všemi závaznými i nepovinnými testy. Pro zvládnutí bonusových testů není tato dodaná funkce použitelná. Jednak v bonusových testech úmyslně počítá nesprávné výsledky a dále by v bonusových testech bylo vhodné jiné rozhraní funkce.

Vaše řešení spočívá v implementaci třídy `CWeldingCompany` se zadaným rozhraním. K této vlastní třídě si můžete přidat další pomocné třídy a funkce, které při implementaci využijete. Rozhraní třídy `CWeldingCompany` používá některé datové struktury, které jsou implementované v testovacím prostředí (jejich zjednodušená implementace je součástí dodaného archivu). Jedná se o třídy `CProd`, `CPriceList`, `COrder`, `COrderList`, `CProducer` a `CCustomer`. Tyto třídy mají implementaci pevně danou, do jejich rozhraní ani implementace nesmíte zasahovat.

V programu je potřeba věnovat péči návrhu vláken. Hlavní vlákno vytvoří instanci `CWeldingCompany`, vyplní zákazníky a dodavatele a spustí vlastní výpočet metodou `Start`. Voláním této metody vzniknou Vaše vlákna obsluhující zákazníky (každý zákazník bude mít své jedno obslužné vlákno) a vlákna pracovní, která budou provádět většinu výpočetně náročných úloh (zejména vlastní naceňování). Obslužná vlákna pro zákazníky a pracovní vlákna bude vytvářet a ukončovat Vaše implementace. Počet pracovních vláken bude předán jako parametr při volání metody `CWeldingCompany::Start`, počet obslužných vláken zákazníků je dán počtem registrovaných zákazníků. V programu dále mohou existovat další vlákna obsluhující odpovědi dodavatelů. Tato vlákna vytváří a ukončuje testovací prostředí. Vlákna dodavatelů s Vaším kódem pracují pouze v okamžiku volání metody `CWeldingCompany::AddPriceList`, kdy předávají

ceník dodavatele. Vlákna dodavatelů jsou opět pouze pomocná, neměly by v nich probíhat žádné intenzivní výpočty.

Po návratu z metody `CWeldingCompany::Start` může hlavní vlákno dělat libovolné činnosti. Vámi vytvořená vlákna v té době obsluhují zákazníky, vyplňují poptávky a komunikují s dodavateli. Hlavní vlákno nakonec zavolá metodu `CWeldingCompany::Stop`. Vaše implementace počká, dokud zákazníci zadávají své poptávky, tyto poptávky zpracuje a odevzdá zpět zákazníkům. Po obsloužení poslední poptávky ukončí vytvořená vlákna (pomocná pro zákazníky i pracovní) a vrátí se do volajícího.

Třída `CWeldingCompany` zajiťuje naceňování poptávek a řídí práci pracovních a dalších vláken. Její impelmentace je na Vás. Rozhraní musí zahrnovat metody:

`SeqSolve(priceList, order)`
: tato metoda slouží k sekvenčnímu otestování vlastního algoritmu skládání prefabrikátů. Pro zadaný ceník prefabrikátů `priceList` a zadaný poptávaný výrobek `order` vypočte jeho cenu a vyplní odpovídající složku v `COrder::m_Cost`. Metoda musí být implementována. Pokud nefunguje, je řešení bez dalšího testování odmítnuto. Pokud se rozhodnete využít dodanou funkci `ProgtestSolver`, bude se jednat o jednoduchý wrapper, který pouze vhodně předá parametry volání.

`AddProducer (prod)`
: metoda přidá dodavatele `prod` do seznamu dodavatelů. Tohoto a ostatních takto registrovaných dodavatelů se budete dotazovat na jejich ceníky. Metoda `AddProducer` je volaná pouze v době, kdy ještě nebyla spuštěna Vaše vlákna (před voláním metody `CWeldingCompany::Start`).

`AddCustomer (cust)`
: metoda přidá zákazníka do seznamu zákazníků. Tohoto a ostatní takto registrované zákazníky budete obsluhovat (vytvoříte pro ně pomocné vlákno, budete přebírat jejich poptávky a budete je naceněné vracet). Metoda je volaná pouze v době, kdy ještě nebyla spuštěna Vaše vlákna (před voláním metody `Start`).

`AddPriceList (prod, priceList)`
: metoda je volaná dodavatelem `prod`, který takto předává svůj ceník `priceList`. Doručení ceníku probíhá buď synchronně nebo asynchronně. Dodavatel obdrží požadavek na předání svého ceníku (nejspíše z nějakého Vašeho pracovního vlákna). Dodavatel tento požadavek buď zpracuje rovnou a `AddPriceList` zavolá přímo z metody žádosti o ceník, nebo asynchronně, tedy metodu `AddPriceList` zavolá později, typicky z jiného vlákna.

`Start(thrCnt)`
: metoda spustí vlákna, zahájí zjišťování poptávek zákazníků, dotazování dodavatelů a vyplňování poptávek zákazníků. Parametrem je celé číslo `thrCnt`, které udává počet vytvářených pracovních vláken. Metoda nastartuje potřebná vlákna (pracovní, obsluhu zákazníků) a vrátí se zpět do volajícího. Metoda nečeká na doběhnutí vytvořených vláken.

`Stop()`
: metoda je volaná z hlavního vlákna, aby ukončila probíhající výpočet. Výpočet není ukončen okamžitě. Metoda počká na obsloužení všech zákazníků a dále počká, dokud nejsou naceněné a vrácené všechny poptávky. Následně metoda `Stop` zařídí ukončení pracovních vláken a pomocných obslužných vláken zákazníků. Ve chvíli, kdy již žádná Vámi vytvořená vlákna neběží, se metoda vrátí zpět do volajícího. Pozor - neukončujte v této chvíli program voláním funkce `exit`, `pthread_exit` nebo podobné. Pokud se volání `Stop` nevrátí do volajícího, bude Váš program označen za nefunkční.

další

do třídy `CWeldingCompany` si můžete doplnit další pomocné metody a členské proměnné, které Vaše implementace použije.

Třída `CProd` představuje jednu položku ceníku dodavatele. Má následující rozhraní:

`konstruktor(w, h, cost)`
: slouží pro snadnou inicializaci členských proměnných,

`m_W, m_H`
: jsou rozměry vyráběného prefabrikátu. Plechy mají stejné vlastnosti v obou směrech, tedy není potřeba rozlišovat šířku a výšku. Například plech s šířkou m_W=3 a výškou m_H=4 lze použít jak na šířku (3 x 4), tak na výšku (4 x 3),

`m_Cost`
: je cena tohoto prefabrikátu.

Třída `CPriceList` představuje celý ceník dodavatele pro daný druh materiálu `materialID`). Má následující rozhraní:

`konstruktor(materialID)`
: slouží pro snadnou inicializaci členských proměnných,

`Add(prod)`
: metoda slouží pro snadné přidávání prefabrikátů do ceníku,

`m_List`
: obsahuje seznam všech prefabrikátů, které dodavatel nabízí pro tento `materialID`.

Třída `COrder` představuje jeden výrobek poptávaný zákazníkem. Má následující rozhraní:

`konstruktor(w, h, weldingStrength)`
: slouží pro snadnou inicializaci členských proměnných,

`m_W, m_H`
: obsahuje výšku a šířku požadovaného výrobku,

`m_WeldingStrength`
: udává požadovanou kvalitu svárů. Pro výpočet tato hodnota vystupuje jako cena za jednotku délky sváru,

`m_Cost`
: představuje cenu, za kterou lze nejméně tento výrobek vyrobit z nabízených prefabrikátů při dodržení ceny sváření. Složka `m_Cost` je inicializovaná na hodnotu 0, během nacenění poptávky je potřeba složku `m_Cost` nastavit na správnou hodnotu.

Třída `COrderList` představuje jednu poptávku od zákazníka. Poptávka je tvořena jedním nebo více poptávanými výrobky:

`konstruktor(materialID)`
: slouží pro snadnou inicializaci členských proměnných,

`Add(order)`
: metoda slouží pro snadné přidávání požadovaných výrobků do poptávky,

`m_MaterialID`
: identifikace materiálu, ze kterého mají být poptávané výrobky vyrobeny,

`m_List`
: seznam poptávaných výrobků, je potřeba nacenit každý z nich.

Třída `CProducer` představuje rozhraní dodavatele. Vlastní implementace dodavatele v testovacím prostředí je podtřídou `CProducer`. Rozhraní je:

`SendPriceList(materialID)`

zavoláním této metody žádáte dodavatele, aby předal ceník svých prefabrikátů pro materiál `materialID`. Volání `SendPriceList` nevrátí požadovaný ceník přímo. Vlastní předání ceníku může proběhnout dvěma způsoby:

- synchronně. Metoda `SendPriceList` v dodavateli připraví ceník a rovnou zavolá Vaši `CWeldingCompany::AddPriceList`. Pak se z `SendPriceList` vrátí.
- asynchronně. Metoda `SendPriceList` v dodavateli pouze zaznamená požadavek na ceník a vrátí se prakticky okamžitě. Vlastní předání požadovaného ceníku proběhne prostřednictvím metody `CWeldingCompany::AddPriceList`, někdy později, pravděpodobně z jiného vlákna. Je na Vaší implementaci, aby dodaný ceník správně převzala, zpracovala a spustila následný výpočet, který je na znalosti ceníku závislý.

Dodavatele se lze dotazovat na stejný `materialID` opakovaně, vracená informace bude vždy stejná.

Třída `CCustomer` představuje rozhraní zákazníka. Vlastní implementace zákazníka v testovacím prostředí je podtřídou `CCustomer`. Rozhraní je:

`WaitForDemand()`

tuto metodu bude v cyklu volat pomocné vlákno pro obsluhu zákazníka. Metoda vrátí smart pointer na instanci `COrderList` - tedy poptávku od zákazníka. Pomocné vlákno poptávku převezme, ale vlastním naceněním poptávky se nezabývá. Pouze ji předá vláknům pracovním k vyřízení. Pomocné vlákno následně opět zavolá `WaitForDemand()`, ve které čeká na další požadavek od zákazníka. Pokud volání `WaitForDemand()` vrátí prázdný smart pointer, znamená to, že zákazník již nemá další požadavky. Pomocné vlákno tohoto zákazníka se v této chvíli může ukončit. Neznamená to ale konec práce programu. Je ještě potřeba dokončit a předat rozpracované poptávky, které zákazník zadal v minulosti.

`Completed(orderList)`

tuto metodu bude volat pracovní vlákno v okamžiku, kdy dokončí nacenění poptávky zákazníka a vyplněnou ji předá zpět. Metoda by měla být zavolána právě jednou pro každou zadanou poptávku.

Funkce `ProgtestSolver (orderList, priceList)` je v testovacím prostředí implementovaná funkce, která dokáže nacenit zadanou poptávku `orderList` podle ceníku `priceList`. Funkci můžete volat ze svého programu, nemusíte se zabývat algoritmickou stránkou problému. Funkce je použitelná v povinných a nepovinných testech. V bonusových testech je funkce úmyslně upravena tak, aby naceňovala špatně. Pokud budete chtít zvládnout i bonusové testy, budete si muset vyvinout vlastní řešení. V přiloženém archivu naleznete hlavičkový soubor `progtest_solver.h` a knihovnu `progtest_solver.a` s implementací této funkce pro lokální testování. Knihovna je zkompilovaná pro Debian Linux 9 (amd64), ale zřejmě bude použitelná i v jiných Linuxových distribucích.

Funkce neřeší konzistenci ceníku. Pokud se v ceníku objeví nekonzistentní údaje, funkce poskytuje nekonzistentní výstup. Nekonzistentním ceníkem se rozumí zejména duplicity - např. následující údaje jsou duplicitní:

```
W  H  Cost
1  1  10
1  1  20
```

Kolidující hodnoty nemusí být takto zřejmé. Protože lze plechy otáčet, jsou duplicitní i případy typu:

```
W  H  Cost
5  7  10
7  5  20
```

Sestavení konzistentního ceníku bez výše uvedených duplicit je na zodpovědnosti volajícího.

Pokud se rozhodnete pro vlastní implementaci, zohledněte následující:

- algoritmus nacenění jednoho poptávaného výrobku by měl mít časovou složitost `O(n m max(n,m)) ∈ O(n^3)`, kde `n x m` je velikost požadovaného výrobku,
- implementace algoritmu by měla být rozumně efektivní, jedná se místo programu, kde se při běhu stráví většina času,
- při testování si testovací prostředí nejprve zkalibruje rychlost Vašeho řešení. Podle naměřené rychlosti pak upravuje velikost zadávaných problémů. Pokud se rychlost Vašeho řešení příliš (řádově) neliší od rychlosti očekávané (referenční), bude Váš program bez problémů otestován. Porovnejte proto rychlost své implementace s rychlostí funkce v dodané knihovně, Vaše implementace by neměla být řádově pomalejší. Při testování jsou zadávané požadavky, kde šířka a výška je v intervalu cca 20 až 200,
- pokud nelze zadaného rozměru dosáhnout (nelze jej poskládat z prefabrikátů v ceníku), je jeho cena nastavena na `DBL_MAX`. Při testování ale bude v ceníku vždy k dispozici prefabrikát velikosti 1x1, tedy vždy půjde sestavit libovolná požadovaná velikost výsledného plechu.

Odevzdávejte zdrojový kód s implementací požadované třídy `CWeldingCompany` 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 6.3, tato verze kompilátoru zvládá většinu C++11 a C++14 konstrukcí.

Doporučení:

- Nemusíte řešit vlastní algoritmus skládání prefabrikátů do požadované velikosti. V prvních řešeních můžete využít nabízenou funkci `ProgtestSolver`, odladit vlákna a synchronizaci. Teprve pak můžete začít s implementací vlastního řešení skládání.

- Abyste zapojili co nejvíce jader, zpracovávejte co nejvíce poptávek od zákazníků najednou. Vyzvedněte je pomocí opakovaného volání `WaitForDemand` jednotlivých zákazníků, zprovozněte komunikaci mezi přebíracími a pracovními vlákny. Není potřeba dodržovat pořadí při vracení vyplněných poptávek. Pokud budete najednou zpracovávat pouze jednu poptávku, nejspíše zaměstnáte pouze jedno vlákno a ostatní vlákna budou čekat bez užitku.
- Instance `CWeldingCompany` 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 tříd poptávek a ceníků 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 alokoval.
- Poptávky musíte načítat, zpracovávat a odevzdávat průběžně. Postup, kdy si všechny poptávky načtete do paměťových struktur a teprve pak je začnete zpracovávat, nebude fungovat. Takové řešení skončí deadlockem v prvním testu s více vlákny. Musíte zároveň obsluhovat požadavky od všech registrovaných zákazníků. Pokud se budete snažit nejprve obsloužit zákazníka A, následně pouze zákazníka B, ..., skončíte taktéž v deadlocku.
- Volání metod `Completed` je reentrantní. Není potřeba jej obalovat mutexem, odevzdávat vyplněnou nabídku může libovolné pracovní vlákno.
- 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 500 MiB 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).

Co znamenají jednotlivé testy:

Test algoritmu (sekvencni)
: Testovací prostředí opakovaně volá metody `SeqSolve()` pro různé vstupy a kontroluje vypočtené výsledky. Slouží pro otestování implementace Vašeho algoritmu. Není vytvářena instance `CWeldingCompany` a není volaná metoda `Start`. Na tomto testu můžete ověřit, zda Vaše implementace algoritmu je dostatečně rychlá. Testují se náhodně generované problémy, nejedná se o data z dodané ukázky.

Základní test
: Testovací prostředí vytváří instanci `CWeldingCmpany` pro různý počet pracovních vláken, dodavatelů a zákazníků. Ve jménu testu je pak uvedeno, kolik je pracovních vláken (W=xxx), dodavatelů (P=xxx) a zákazníků (C=xxx).

Test zahlcení
: Testovací prostředí generuje velké množství požadavků a kontroluje, zda si s tím Vaše implementace poradí. Pokud nebudete rozumně řídit počet rozpracovaných požadavků, překročíte paměťový limit.

Test zrychleni vypoctu
: Testovací prostředí spouští Vaši implementaci pro ta samá vstupní data s různým počtem pracovních vláken. Měří se čas běhu (wall i CPU). S rostoucím počtem vláken by měl wall time klesat, CPU time mírně růst (vlákna mají možnost běžet na dalších CPU). Pokud wall time neklesne, nebo klesne málo (např. pro 2 vlákna by měl ideálně klesnout na 0.5, je ponechaná určitá rezerva), test není splněn.

Busy waiting (pomali zakaznici)
: Do volání `WaitForDemand` testovací prostředí vkládá uspávání vlákna (např. na 100 ms). Výpočetní vlákna tím nemají práci. Pokud výpočetní vlákna nejsou synchronizovaná blokujícím způsobem, výrazně vzroste CPU time a test selže.

Busy waiting (pomali dodavatele)
: Reakce dodavatelů na požadavek o ceník je uměle prodloužena (odložena např. o 100ms). Výpočetní vlákna tím nemají práci. Pokud výpočetní vlákna nejsou synchronizovaná blokujícím způsobem, výrazně vzroste CPU time a test selže.

Busy waiting (pomale akceptace)
: Do volání `Completed` je vložena umělá prodleva. Pokud vlákna obsluhující zákazníky nejsou nejsou synchronizovaná blokujícím způsobem, výrazně vzroste CPU time a test selže (ale tento scénář není příliš pravděpodobný).

Rozlozeni zateze 1
: Testovací prostředí zkouší, zda se do řešení jedné poptávky dokáže zapojit více pracovních vláken. Pokud chcete v tomto testu uspět, musíte Váš program navrhnout tak, aby bylo možné využít více vláken i při zpracování jediné poptávky zákazníka, kde je poptáváno více různých výrobků. Jedná se o test bonusový.

Rozlozeni zateze 2
: Testovací prostředí zkouší, zda se do řešení jedné poptávky dokáže zapojit více pracovních vláken. Pokud chcete v tomto testu uspět, musíte Váš program navrhnout tak, aby bylo možné využít více vláken i při zpracování jediné poptávky jediného zákazníka, kde je poptávaný jeden velký výrobek. Jedná se o test bonusový.