Dokumentace

Implementace diskrétního simulátoru s podporou SHO

IMS - Modelování a simulace

Iveta Strnadová
xstrna14
Denis Lebó
xlebod00

7. prosinec 2020

Obsah

1. Úvod

Práce se zabývá vývojem simulátoru založeného na událostech s podporou systému hromadné obsluhy. Tento systém popisuje, vysvětluje pro použití a na zvolené simulaci ho implementuje.

Zadání

Téma č. 4: Implementace diskrétního simulátoru s podporou SHO

Implementujte vlastní diskrétní simulátor založený na procesech nebo událostech. Implementujte podporu pro SHO (fronty, linky) a generování pseudonáhodných čísel (pro různá rozložení). Demonstrujte na hypotetickém modelu SHO s několika linkami, různými procesy příchodů apod.

Pojmy

Kvůli implementaci simulátoru v anglickém jazyce (pro snazší porovnání s ostatními simulátory, např. SIMLIB, kterým byly názvy tříd a funkcí částečně inspirovány) jsou v této práci používány pro popis české a u konkrétní implementace anglické výrazy. Pro snazší zorientování čtenáře jsou zde vypsány ty nejpoužívanější a jejich ekvivalenty v druhém jazyce.

event = událost
seize = zabrat
release = uvolnit
storage = sklad
zařízení = facility

2. Simulátor

Stavba

Jeden hlavičkový soubor `discrete_simulator.hpp` obsahuje deklarace všech tříd a funkcí. Definice jsou umístěny v několika souborech podle funkce.

Simulátor (`simulator.cpp`)

Statická implementace třídy `Simulator` funguje jako hlavní rozhraní mezi uživatelem a simulátorem. Zajišťuje inicializaci simulátoru a jeho spuštění, tj. procházení naplánovaných eventů a jejich vykonávání a obsahuje funkci pro naplánování eventů.

Procházení eventů, akce po spuštění simulace příkazem `Run()`, probíhá vybíráním prvního eventu z jejich seznamu a spuštěním jeho funkce `Behaviour()`. Při vkládání eventu je zařazen podle času a shodují-li se časy, ten s vyšší prioritou má přednost.

Simulátor také obsahuje mapy všech vytvořených zařízení a skladů, které uživateli umožňuje vytvářet a které se mohou eventy pokusit zabrat a následně opustit. Při pokusu o zabrání zařízení či skladu navíc umožňuje nastavit timeout pro čekání ve frontě při nedostatku kapacity.

Eventy (`events.cpp`)

Existují zde tři druhy tříd eventů.

Třída `Event` je základem všech eventů. Pro vytvoření vlastního eventu uživateli stačí vytvořit novou třídu dědící z Event a definovat její `Behaviour()`, tedy funkci, která se provede v momentě, kdy je event vybrán z naplánovaných a spuštěn.

Třída `EventGenerator` existuje stejně jako Event jako rodičovská třída, z níž si uživatel vytvoří vlastní. Oproti Event obsahuje navíc proměnnou, která zajistí, že se po provedení jejího `Behaviour()` v simulátoru neodstraní.

Třída `TimeoutWatch` není přístupná pro uživatele. V případě, že uživatel bude chtít zabrat sklad/zařízení s timeoutem, naplánuje se po daném čase a pokud najde sledovaný event stále čekající ve frontě, vyjme ho a spustí event předaný uživatelem pro případ timeoutu.

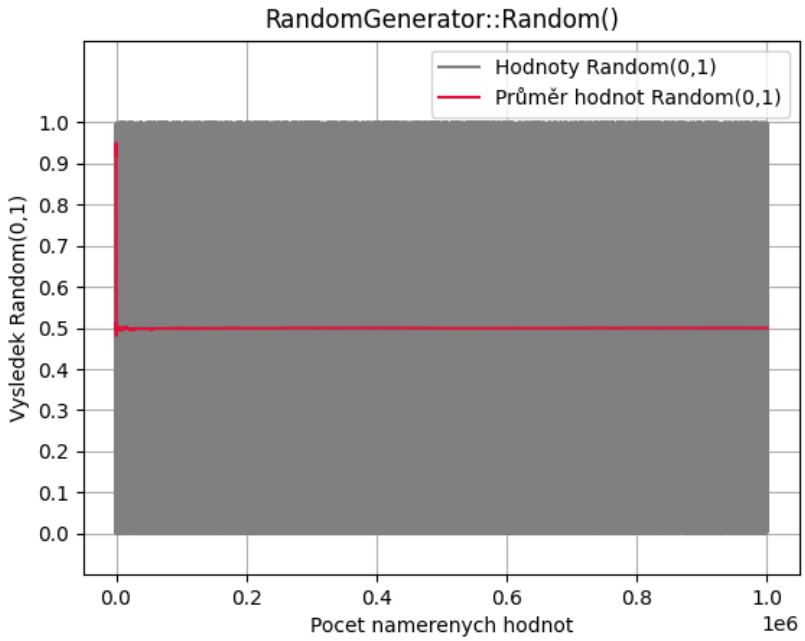
Generátor náhodných rozložení (`random_generator.cpp`)

Pseudo generátor náhodných rozložení pro svou funkci využívá náhodného seed, které se vygeneruje při inicializaci simulátoru a vypíše na standartní výstup. Uživatel může tuto hodnotu změnit na vlastní.

Generátor obsahuje funkci pro generování následujících náhodných čísel:

Random - náhodné od 0 do 1

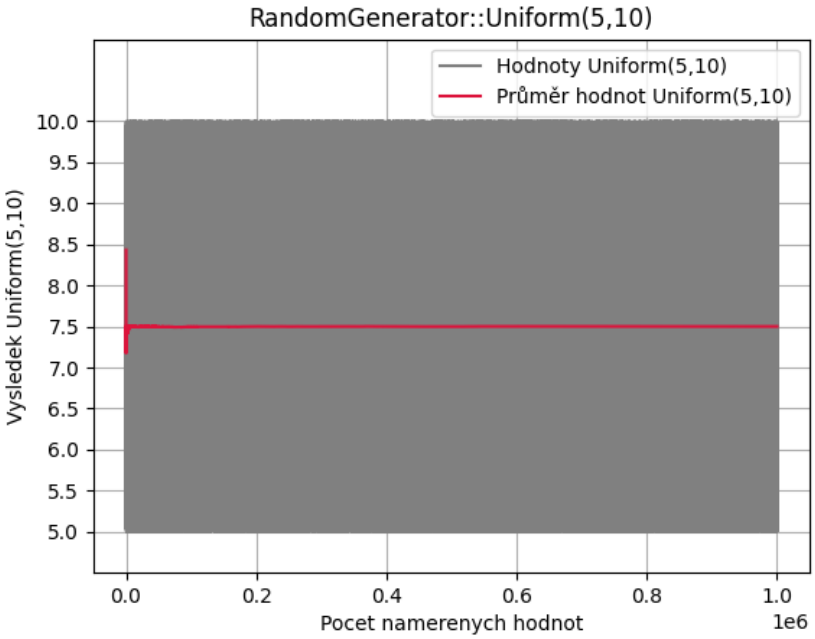
Náhodně vygenerované 32 bitové číslo `uint32_t randomNumber` je vygenerované pomocí `RandomNumberGenerator()`, poté je vyděleno maximálním číslem pro rozsah uint32_t plus 1.0. Tato matematická operace vrací číslo z oboru [0,1). Jeho hodnota je převedena na hodnotu typu double a je vrácena.
Níže je uveden graf s naměřenými hodnotami a vývojem střední hodnoty naměřených hodnot - počet měření byl jeden milion a počáteční hodnota `randomNumber` byla 0x626f6f70.



Uniformní

Podobně jak ve funkci `Random()` je číslo vygenerované pomocí `RandomNumberGenerator()` a je vyděleno stejnou hodnotou jak ve funkci `Random()`, avšak hodnota `randomNumber` je vynásobena rozdílem mezi vrchní a spodní hranicí uniformního rozdelení, což vrátí hodnotu v oboru [0, MAX) , kde MAX je vrchní hranicí uniformního rozdelení. Poté je k tomuto výsledku připočítána spodní hranice rozdělení pro získání čísla z výsledného oboru [MIN, MAX), kde MIN je spodni hranice a MAX je vrchní hranice rozdělení.

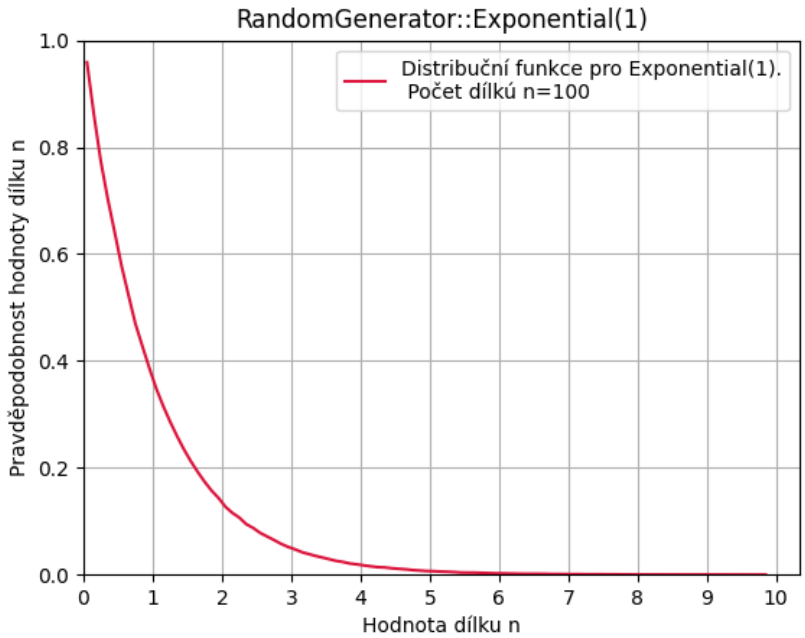
Níže je uveden graf s naměřenými hodnotami a vývojem střední hodnoty naměřených hodnot - počet měření byl jeden milion, počáteční hodnota `randomNumber` byla 0x626f6f70, spodní hranice byla 5 a vrchní 10.



Exponenciální

Generovali jsme hodnoty s exponenciálním rozdělením pomoci vzorce $T = F^{-1}_X(U)$, kde F^{-1}_X je inverzní distribuční funkce pro exponenciální rozdelení a U je hodnota z rovnoměrného rozdelení z oboru [0,1]. Tento výpočet vychází z metody inverzní transformace. Pokud X je náhodnou hodnotou ze spojité množiny pro kterou existuje distribuční funkce, má potom náhodná proměnná $U = F_X(X)$ rovnoměrné rozložení na oboru [0,1]. Takže jestli U má náhodní rovnoměrné rozložení na oboru [0, 1] a X má distribuční funkci F_X, má potom náhodná proměnná $T = F^{-1}_X(U)$ stejné rozložení jako X. Tímto procesem dospějeme ke vzorci T = E(X) * log(U), kde U je hodnota získaná za pomoci funkce `Random()`.

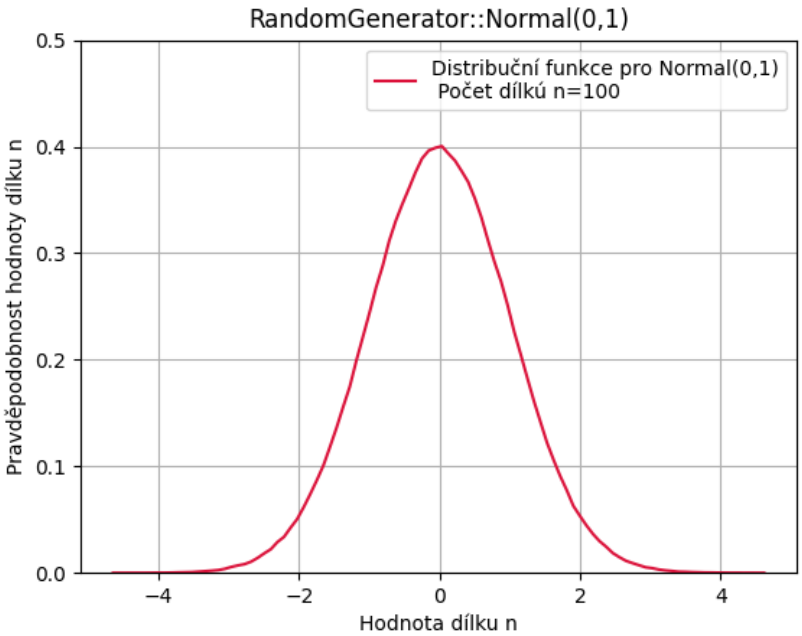
Níže je uveden histogram pro naměřené hodnoty z funkce `Exponencial(1)`, počet dílků je 100 pro obor [0,10] - počet měření byl jeden milion, počáteční hodnota `randomNumber` byla 0x626f6f70.



Normální

Generovali jsme hodnoty s normálním rozdělením pomocí Box-Mullerovi transformace, kde jsou vygenerovány dvě čisla `z0` a `z1`, které jsou nezávislými čísly s normálním rozdelením se střední hodnotou `mu` a rozptylem `sigma`. Je navráceno jedno z těchto čísel a druhé je uloženo pro další volání funkce. Tudíž funkce je nucena provádet výpočet pouze při každém lichém volání funkce, nebo při změně `mu` anebo `sigma` mezi voláními.

Níže je uvedem histogram pro naměřené hodnoty z funkce `Normal(0,1)`, počet dílků je 100 pro celý obor hodnot - počet měření byl jeden milion, počáteční hodnota `randomNumber` byla 0x626f6f70.



Systém hromadné obsluhy (`sho.cpp`)

Obsahuje třídy pro podporu SHO. Všechny jsou uživateli nepřístupné, vytváří a používá je pouze pomocí funkcí v simulátoru k tomu určené.

`Queue` implementuje frontu čekajících eventů v případě plné kapacity zařízení/skladu Může mít omezený limit nebo nemusí (popř. může být nulová). Je-li při novém požadavku na zabrání plná, uživateli se o tom vrací informace. Příchozí eventy řadí podle priority, vyšší priorita má přednost.

`Storage` / `Facility` jsou podobné třídy. Jediný rozdíl je v kapacitě, zatímco sklad může mít libovolnou, zařízení má vždy jen jednu. Obě třídy sledují volnou kapacitu. Spouští eventy, které se je pokusí zabrat pokud je kapacita volná, nebo je ukládají do fronty když kapacita volná není. Při uvolnění jejich kapacity spouští event, který je vpředu fronty.

Jsou rozpoznávány pomocí jejich jména. Musí být definovány před spuštěním simulace a pokud dojde k redefinici nebo použití nedefinovaného skladu/zařízení, program se ukončí.

Informační výpisy (`statistics.cpp`)

Obsahuje statickou třídu `Log` obstarávající výpisy o stavu simulátoru a o aktuálně spouštěném eventu.

Návod k použití

Připojení simulátoru a nezbytné funkce

```
#include "discrete_simulator.hpp"
int main()
{
    double start_time = 0.0;
    double end_time = 10.0;
    Simulator::Init(start_time, end_time);
    Simulator::Run();
}
```

`Simulator::Init(start_time, end_time)` - inicializuje simulátor a nastaví čas simulace.
`Simulator::Run()` - spustí simulaci podle toho, co bylo definováno mezi `Init` a `Run`. Je důležité, aby případné další funkce byly volané mezi těmito funkcemi.

Vytváření a plánování eventů

```
class Person : public Event {
    public:
        void Behaviour() {
            Log::EventState(this, "Person appeared in the system.")
        }
};
class PersonGenerator : public EventGenerator {
    public:
        void Behaviour() {
            double next_time = Simulator::last_effective_time + 1;
            Simulator::ScheduleEvent(new Person(), next_time);
            Simulator::ScheduleEvent(this, next_time);
        }
}
int main() {
    Simulator::Init(0.0, 2.0);
    Simulator::ScheduleEvent(new PersonGenerator(), 0.0);
    Simulator::Run();
}
//výstup:
//[1]   Event#1: Person appeared in the system.
//[2]   Event#2: Person appeared in the system.
```

Třída pro vlastní event i generátor se vytvoří děděním z `Event` / `EventGenerator` a napsáním vlastního `Behaviour`. V tomto případě generátor vytvoří event Person a naplánuje sebe i jeho do kalendáře událostí za 1 (např. 1 sekundu, záleží jak se na čas díváme).

Proměnná `Simulator::last_effective_time` v sobě uchovává čas posledního spuštěného eventu.

`Simulator::ScheduleEvent(*event, time)` - naplánuje událost event na čas simulace time

Další funkce simulátoru

`Simulator::Wait(time_to_wait, *event)` - počká daný čas a poté spustí přiložený event

`Simulator::CreateStorage(name, capacity)` / `Simulator::CreateStorage(name, capacity, queue_limit)` - vytvoří sklad daného jména, kapacity a popř. s omezenou frontou

`Simulator::CreateFacility(name)` / `Simulator::CreateFacility(name, queue_limit)` - vytvoří zařízení daného jména, popř. s omezenou linkou

Následující funkce jsou uvedeny pouze pro `Storage`, protože pro `Facility` fungují úplně stejně.

`Simulator::SeizeStorage(name, *event)` - pokusí se zabrat sklad, daný event se vykoná až se dostane na řadu, vrací `false` u plné fronty

`Simulator::SeizeStorageWithTimeout(name, timeout_time, *on_success_event, *on_timeout_event)` - pokusí se zabrat sklad; pokud zůstane ve frontě a bude tam i po uplynutí timeout_time, on_success_event se z ní vyjme, zahodí a provede se místo něj on_timeout_event

`Simulator::ReleaseStorage(name)` - uvolní jednotku ze skladu, pokud byl ve frontě event, umístí se jako příští provedený

Poznámka

Simulátor založený na událostech (eventech) je v určitých ohledech omezující. Funkce, které vyžadují v sobě odkaz na event(y) vyžadují speciální zacházení (tj. funkce `Wait`, `SeizeStorage/Facility` a `SeizeStorage/FacilityWithTimeout`).

Protože jejich roli v simulaci "přebírají" přiložené eventy, které se mohou vykonat ihned nebo až po chvíli, event který tyto funkce volá ve svém `Behaviour` již nesmí provést žádnou jinou funkci. Jedinou výjimkou je zpracování hodnoty false, kterou mohou funkce `Seize` vrátit, protože v tom případě se přiložený event nikdy neprovede.

Funkce z náhodného generátoru čísel

`RandomGenerator::SetSeed(seed)` - nastaví hodnotu seed na požadovanou hodnotu

`RandomGenerator::Random()` - vrátí náhodnou hodnotu z intervalu [0, 1) s rovnoměrným rozdělením

`RandomGenerator::Uniform(low, high)` - vrátí náhodnou hodnotu z intervalu [low, high) s rovnoměrným rozdělením

`RandomGenerator::Exponential(mean)` - vrátí náhodnou hodnotu z exponenciálního rozložení se středem v mean

`RandomGenerator::Normal(mean, std_deviation)` - vrátí náhodnou hodnotu z normálního rozložení se středem v mean a standartní odchylkou std_devition

Poznámka: RandomGenerator neošetřuje nevhodné záporné vstupy či výstupy z funkcí generujících náhodná rozložení. Je na uživateli, aby zvážil co generuje a ošetřil případné mezní případy.

Funkce z `Log`

`Log::SimulatorState` - vypíše na stdout základní informace o simulátoru: čas, počet naplánovaných eventů a eventů pozastavených ve frontách, počet definovaných skladů/zařízení a podrobnější informace o nich

`Log::EventState(*event, msg)` - vypíše na stdout čas simulace, jméno nebo identifikátor předané události a danou zprávu msg

Podrobnější popis zde zmíněných funkcí, jejich parametrů a efektů na simulaci můžete najít v hlavičkovém souboru simulátoru (`discrete_simulator.hpp`). Ukázku použití prvků simulátoru najdete v implementaci simulace (`simulation.cpp`).

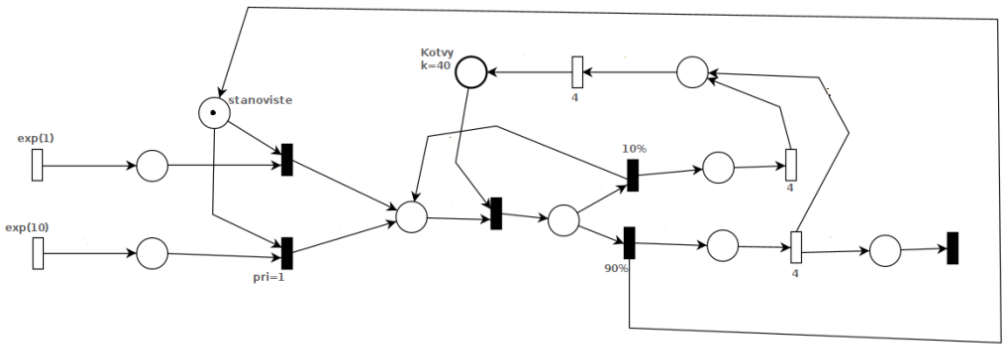
3. Simulace

Příklad "vlek":

- Lyžaři: obyčejní - exp(1), závodníci - exp(10), závodníci mají přednost
- 40 kotev, jedno startovní stanovište
- při nastupování:
    - 10% - chyba, nástup se opkauje, kotva jede dál
    - 90% - 4 min nahoru, kotva jede další 4 min zpět

Příklad byl převzat z prvního democvičení v předmětu IMS - Modelování a simulace, kde byl použit jako ukázka modelování petriho sítí.

Abstraktní model



Implementace simulace

Podařilo se nám implementovat abstraktní model Petriho sítě se všemi charakteristickými vlastnostmi modelu.

Jednotlivé implementované stavy jsou deklarovány na začátku `simulation.cpp` souboru a dědí své vlastnosti od třídy Event. Kvůli omezení z povahy simulátoru založeného na událostech (eventech) jsou závodníci a lyžaři implementování několika eventy, které se postupně navzájem vytvářejí.

Její chování je zavedeno pomocí funkcí SeizeFacility, SeizeStorage, Wait, ScheduleEvent, ReleaseFacility a ReleaseStorage, jejichž konkrétní implementace lze najít ve funkci `Behaviour()` pro každou třídu. Stavy `kotva` a `stanoviste` byly zavedeny pomocí CreateStorage s kapacitou 40 a CreateFacility respektivne, obě byli zavedeny bez omezení na délku fronty. Zdroje pro generování lyžařů a závodníků byly implementovány pomocí třídy `EventGenerator` ze simulátoru a délky mezi jejich opakováním jsou dány návratovou hodnotou funkcí `Exponencial(1.0)` pro lyžaře a `Exponencial(10.0)` pro závodníky.

Po počátečním testování jsme změnili velikost skladu kotev na 2, aby se vytvořily fronty před zařízením `stanoviste` a projevila se tak lépe funkce priorit v systému, kdy závodník dostane přednost před lyžařem.

4. Závěr

Přiložené zdrojové kódy simulátoru se dají použít jako primitivní simulační knihovna založená na událostech. Potenciální uživatel může kromě této dokumentace využít k nalezení více informací hlavičkový zdrojový soubor či vytvořený program simulace obsahující většinu výše zmíněných konstrukcí.