Implementace diskrétního simulátoru založeného na přepínání procesů

Poláček Marek
xpolac06@stud.fit.vutbr.cz

Mikulka Jiří
xmikul39@stud.fit.vutbr.cz

Vysoké učení technické v Brně
Fakulta informačních technologií

17. prosince 2010

1 Úvod

Modelování [IMS, slide 8] je činnost, která vede k získávání informací o jednom systému [IMS, slide 7] prostřednictvím jiného systému (modelu [IMS, slide 7]). Podle prvků systému je možné systémy rozdělit do několika skupin [IMS, slide 32]:

diskrétní systém je takový systém, ve kterém mají všechny prvky diskrétní chování (tzn. jejich stav se mění skokově)

spojitý systém je takový systém, ve kterém mají všechny prvky spojité chování (tzn. jejich stav se nemění skokově, ale spojitě)

kombinovaný systém je takový systém, ve kterém se vyskytují jak prvky s diskrétním, tak se spojitým chováním

Aby bylo možné získávat znalosti o skutečném systému, je nutné vytvořit jeho abstraktní model. Abstraktní model je účelový a zjednodušený popis zkoumaného systému. Proces převodu abstraktního modelu do programu znamená vytvoření simulačního modelu. [IMS, slide 10]

Knihovna funkcí, která byla v rámci tohoto projektu implementována, umožňuje uživatelům vytváření simulačních modelů diskrétních systémů (tedy formou programu zapsat chování reálného diskrétního systému).

Při zkoumání chování reálných systému (např. samoobsluhy, automatu na jízdenky, vlakového nádraží aj.) není možné postihnout všechny možné stavy systému pouhým pozorováním. Proto je vhodné použít počítač, na kterém je spuštěn simulační model, který provádí náhodné experimenty s abstraktním modelem. Výsledkem těchto experimentů jsou pak nové znalosti o systému. Tyto výsledky je nutné vhodně interpretovat a ověřit, zda opravdu vypovídají o modelovaném systému.

V této práci je popsána implementace diskrétního simulátoru založeného na přepínání procesů. Chování jednotlivých objektů v systému bude tedy popsáno procesy (ty jsou implementovány pomocí vláken). Základní principy činnosti toho simulátoru včetně používaných součástí (zařízení, sklad, proces) jsou popsány v [IMS, slide 124 - 208]. V tomto textu bude odkazováno do této části, neboť celkově pokrývá přístup k distrétní simulaci.

Pro ověření správnosti implementované knihovny byly vytvořeny programy pro testování knihovny. Tyto experimenty budou popsány v kapitole 5; jejich smyslem nebylo modelování konkrétního reálného systému a provádění experimentů nad modelem, ale ověření funkčnosti implementované knihovny.

Přestože v [IMS, slide 124 - 208] je popsáno, ze kterých částí se sestává diskrétní simulátor a jak diskrétní simulace probíhá [IMS, slide 176], bylo nutné navrhnout způsob implementace ve zvoleném jazyce[1].

1.1 Zdroje faktů

Problematika diskrétní simulace je podrobně popsána v [IMS, slide 124 - 208] nebo v [2]. Při implementaci knihovny jsme proto vždy vycházeli z principů popsaných v těchto materiálech. Pokud informace v těchto materiálech byly příliš obecné, obraceli jsme se na dokumentaci knihovny SIMLIB[4].

Implementační záležitosti (především problematiku vláken) jsme vyhledávali v manuálových stránkách[3] a v [1].

2 Rozbor tématu a použitých metod/technologií

Diskrétní simulátor je druh simulátoru, ve kterém se objevují pouze prvky s diskrétním chováním (tedy prvky, jejichž stav se mění skokově). Popis chování prvků systému je v diskrétním simulátoru možný pomocí procesů nebo pomocí událostí. Knihovna implementuje způsob popisu chování pomocí procesů (viz zadání[2]). Pomocí diskrétního simulátoru je možné simulovat SHO[3], který obsahuje několik základních prvků [IMS, slide 139]:

[1] Pro implementaci diskrétního simulátoru byl zvolen jazyk C.

[2] `http://perchta.fit.vutbr.cz/vyuka-ims/15`

[3] Systém hromadné obsluhy (angl. Queuing system)

transakce – Transakcemi jsou myšleny **procesy**, které mají specifické chování (tj. popis průchodu procesu celým systémem). Všechny procesy nemusejí být stejné - mohou se lišit jak svým chováním, tak svojí prioritou. Právě priorita je rozhodující při obsazování obslužných linek.

obslužné linky – V SHO je více typů obslužných linek - **zařízení** (obslužná linka s kapacitou 1) a **sklady** (obslužná linka s kapacitou větší nebo rovnou 1). Každé zařízení (resp. sklad) může mít frontu, ve které procesy čekají, dokud nemohou obsadit linku (tj. zařízení nebo sklad).

fronty – Existuje několik druhů **front** [IMS, slide 141], které se v SHO využívají – **frontové řády** (FIFO (fronta), LIFO (zásobník), SIRO (fronta s náhodným pořadím)), **nulová fronta** (procesy nemohou vstupovat do fronty (systém se ztrátami)), **konečná fronta** (kapacita fronty je omezena) a **fronta s netrpělivými požadavky** (netrpělivý požadavek opouští frontu, pokud jeho doba čekání přesáhne určitou mez).

Aby námi implementovaná knihovna bylo schopná simulovat diskrétní systémy založené na přepínání procesů, bylo nutné definovat a vytvořit několik základních tříd [IMS, slide 166]:

- **Process** – báze pro modelování procesů
- **Facility** – obslužná linka s výlučným přístupem
- **Store** – obslužná linka s kapacitou
- **Queue** – fronta
- **Stats** – vytváření statistik obslužných linek

Tyto třídy budou podrobně probírány v kapitole 4.

Řízení celé simulace je prováděno jediným centrálním prvkem, což je **kalendář** [IMS, slide 176], jehož chování lze zapsat takto:

```
< inicializace času, kalendáře >
while ( < kalendář je neprázný > ) {
        < vyjmutí prvního prvku z kalendáře >
        if ( < aktivační čas události je větší než čas konce simulace > )
                < konec simulace >
        < nastavení nového simulačního času >
        < provedení chování vyjmutého prvku >
};
```

2.1 Použité postupy

Pro vypracování knihovny bylo nutné zvolit způsob, jak implementovat provádění procesů. Zvolili jsme vláknový přístup, protože vlákna poskytují odlehčený přístup a celkově menší režii než například řešení pomocí tvoření nových procesů (`fork(), vfork()`. Existuje několik knihoven, které je možno použít pro práci s vlákny. My jsme sáhli po osvědčené NPTL (*Native POSIX Thread Library*) ve verzi 2.12.1. Jiné knihovny typu LinuxThreads nepodávají tak dobré výsledky.

V knihovně existuje mnoho situací, kdy je nezbytné uspat právě provádějící proces, dát možnost bežet procesu jinému a po splnění nějaké podmínky znovu pustit proces první. K tomuto účelu je výhodné použít funkce jako jsou `pthread_cond_signal()`, `pthread_cond_broadcast()` či `pthread_cond_wait()`. Dále je nutné ovládat základní práci s mutexy.

Jelikož je pochopení práce s těmito prostředky pro vývoj knihovny zásadní, je žádoucí si jejich použití rozebrat a vysvětlit. Ke snažšímu pochopení slouží následující příklad:

```
1  #include <error.h>
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
```

```
 7  static pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
 8  static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
 9
10  static void *foo(void *p)
11  {
12          pthread_mutex_lock(&mut);
13
14          /* ... */
15
16          pthread_cond_signal(&cond);
17          pthread_mutex_unlock(&mut);
18
19          return NULL;
20  }
21
22  int main(void)
23  {
24          pthread_t th;
25          pthread_mutex_lock(&mut);
26          pthread_create(&th, NULL, foo, NULL);
27          pthread_cond_wait(&cond, &mut);
28          pthread_join(th, NULL);
29
30          return EXIT_SUCCESS;
31  }
```

Nejprve se ve funkci `main()` zamče mutex. To je duležité proto, protože chceme mít deterministické chování vláken. V momentě, kdy se vytvoří vlákno voláním `pthread_create()`, nemáme zaručeno, kdo skutečně poběží jako první. To záleží na plánovači operačního systému. My požadujeme, aby jsme se nejprve dostali k čekání na signál (funkce `pthread_cond_wait()`). I kdyby nejprve běželo nově vytvořené vlákno, víme, že se zastaví na zamknutí mutexu ve funkci `tf()` a bude na tomto místě čekat, dokud druhé vlákno mutex neodemče. Jakmile se dostaneme k čekání `pthread_cond_wait()`, tak funkce způsobí to, že se onen mutex odemče a začne se čekat na signál. Toto odemknutí dá šanci, aby se provedlo tělo funkce `tf()`. Tam dojde k úspěšnému zamčení mutexu a odeslání signálu na condition variable. Nicméně v tomto momentu ještě nemůže program pokračovat ve funkci `main()`, protože funkce `pthread_cond_wait()` před svým návratem potřebuje daný mutex zamknout. Toto bude možné až v momentě, kdy se provede odemčení mutexu ve funkci `tf()`. Poté se bude pokračovat ve vykonávání funkce `main()`, kde je již jen čekání na dokončení provádění vlákna a odebrání jeho návratové hodnoty pomocí funkce `pthread_join()`. Poté se ukončí úspěšně celý proces. Je nutné si uvědomit, že kdyby se nejprve provedla celá funkce `tf()` a teprve pak se zahájilo čekání, signál poslaný z `tf()` se ztratí a čekání se tak nikdy neukončí.

3 Koncepce modelu

V této kapitole bude nastíněn návrh knihovny včetně všech důležitých součástí a funkcí. Všechny prvky SHO a poskytované funkce vycházejí z [IMS, slide 124 - 208].

3.1 Knihovna funkcí

Zde je uveden přehled hlavních struktur, které jsou v knihovně využívány (buď jako interní nebo externí, tj. přístupné uživateli). U každé struktury jsou uvedeny pouze nejvýznamější funkce včetně jejich významu.

- **Calendar** – implementuje kalendář (tj. hlavní plánovací mechanismus knihovny)
  - `Init` – inicializace kalendáře (čas, ...)
  - `Run` – běh simulace
- **Process** – implementuje proces (tj. transakce se specifickým chováním)

  - `Wait` – provede přeplánování procesu v kalendáři
  - `Quit` – ukončí proces
  - `create_process` – vytvoří proces
- **Facility** – implementuje zařízení (tj. linka s výlučným přístupem)
  - `Seize` – obsazení zařízení
  - `Release` – uvolnění zařízení
  - `fac_set_name` – nastavení jména zařízení
  - `fac_get_name` – získání jména zařízení
  - `fac_queue_len` – získání délky fronty
  - `fac_busy` – zjištění stavu zařízení
- **Store** – implementuje sklad (tj. linka s kapacitou)
  - `Enter` – zabrání kapacity
  - `Leave` – uvolnění kapacity
- **Queue** – implementuje prioritní frontu (tj. řazení podle času příchodu a priority procesu)
  - `pq_push` – vložení prvku do fronty
  - `pq_push_attr` – vložení prvku s atributem
  - `pq_top` – získání prvního prvku
  - `pq_top_attr` – získání atributu prvního prvku
  - `pq_pop` – odstranění prvního prvku
  - `pq_empty` – zjištění prázdnosti fronty
- **Stats** – implementuje statistiky a generátory náhodných čísel
  - `save_time` – vloží časový údaj do statistiky
  - `print_stats` – vytiskne statistiky pro zařízení (resp. sklad)
  - `Exponential` – generuje číslo s exponenciálním rozložením
  - `Random` – generuje náhodné číslo
  - `Uniform` – generuje číslo ze zadaného intervalu
  - `Normal` – generuje číslo s normálním rozložením

Implementace jednotlivých struktur a funkcí bude podrobněji popsána v kapitole 4, kde je vysvětlena i logika a princip jednotlivých struktur (resp. funkcí).

4 Architektura simulačního modelu/simulátoru

Tato kapitola popisuje vlastní implementaci diskrétního simulátoru. Nejprve bude zmíněna celková struktura simulátoru. Poté bude popsán způsob návrhu programu.

4.1 Mapování konceptuálního modelu do simulačního modelu

Knihovna se skládá z několika částí, které jsou schopny spolu kooperovat a komunikovat. Jednotlivé objekty slouží k modelování základních zařízení, tvoření statistik a popis chování procesů. Zaměříme se především na části proces, kalendář, zařízení, sklad, statistiky a frontu.

4.2 Proces

Procesy reprezentují dynamické provádění různých akcí. Proc je posloupnost událostí. [IMS, slide 124]. Každý proces je tvořen strukturou `process_struct`. Tato struktura má tvar:

```
struct process_struct {
        volatile int state;
        int prio;
        double atime;
        pthread_t th;
        pthread_mutex_t lock;
        pthread_cond_t cond;
        void *(*behaviour) (void *);
};
```

Proměnná `state` uchovává stav procesu. Proces může mít 4 základní stavy. Stav `TASK_RUNNING` říká, že bylo již vytvořeno vlákno odpovídající procesu a tento proces právě běží. `TASK_STOPPED` je stav, kdy již bylo vytvořeno odpovídající vlákno, ale vlákno neběží a čeká na povel od kalendáře, aby mohlo znovu běžet. `TASK_WAKING` představuje proces, který má sice inicializovanou svou strukturu, ale ještě v systému nebylo vytvořeno jeho vlákno. Poté, co proces dokončí, se nachází ve stavu `TASK_DEAD`, což symbolizuje to, že se čeká, než si systém odebere návratový kód vlákna. Teprve poté bude vlákno definitivně odstraněno ze systému.

Další prvek struktury procesu je `prio`, což je priorita procesu. Vyšší hodnota této proměnné značí vyšší prioritu. Tato proměnná ovlivňuje řazení procesu do fronty [IMS, slide 181].

Proměnná `atime` nese aktivační čas procesu, to jest simulační čas, kdy se znovu začne tento proces provádět.

Proměnné `th`, `lock` a `cond` jsou nutné pro ovládání každého běžícího vlákna a umožňují jeho pozastavování a opětovné spouštění.

Posledním prvek je proměnná `behaviour`. Tato proměnná má v sobě uloženu adresu funkce, která se bude běžícím vláknem provádět. Na tuto funkci je kladen požadavek mít deklaraci ve tvaru `void *start_routine(void *)`.

Všechny procesy v simulaci jsou seskupeny do pole `process_list`.

Knihovna obsahuje několik funkcí pro manipulaci s procesy. Funkce `create_process()` vytvoří strukturu procesu a inicializuje ji počátečními hodnotami. Také provede vložení prvku do kalendáře.

Funkce `Wait()` provede pozastavevní provádění procesu na dobu specifikovanou parametrem. Opětovné spuštění procesu tedy proběhne v simulačním čase `T + t`, kde `T` je aktuální simulační čas a `t` je hodnota parametru předaného této funkci [IMS, slide 180]. Čas `T + t` se někdy nazývá *reaktivační čas*. Ve své podstatě tato funkce způsobi přeplánování procesu, poté pošle signál kalendáři a uspí aktuální proces. Kalendář obdrží signál a provede spuštění jiného procesu.

Pro ukončení procesu se používá funkce `Quit()`. Tato funkce označí proces jako mrtvý a pošle signál kalendáři. Neprovádí žádné přeplánovávání.

4.3 Kalendář

Kalendář je uspořádaná datová struktura uchovávající aktivační záznamy událostí [IMS, slide 176]. Vždy vybírá jako další proces ten, který má nejmenší aktivační čas. Kalendář je implementovaný jako jednosměrný seznam. V jednom okamžiku běží právě jeden proces [IMS, slide 177]. Tomuto chování říkáme *kvaziparalelismus*. Záznam v kalendáři tvoří struktura

```
struct cal {
        struct cal *next;
        size_t idx;
};
```

Zde `next` je ukazatel na následující prvek v kalendáři a `idx` je index odpovídajícího procesu v seznamu všech procesů, `process_list`.

Kalendář si uchovává tři záznamy o simulačních časech. Prvním z nich je `start_time`, zde je uloženo, kdy simulace započala. Oproti tomu `end_time` je čas, kdy simulace musí skončit. Aktuální simulační čas je uschován v proměnné `cur_time`.

Pro manipulaci s kalendářem slouží funkce `add_elem()`. Tato funkce vkládá záznam do samotného kalendáře. Vkládá ho takovým způsobem, aby byly prvky v kalendáři seřazeny podle různých kritérií. Nejdůležitejším kritériem je aktivační čas, zde platí, že prvek s menším aktivačním časem je vždy před prvkem s větším aktivačním časem. Druhým kritériem je priorita procesu. Pokud dojde k situaci, že se přidává prvek, který má stejný aktivační čas jako některý prvek, co je již v kalendáři, platí, že prvek s vyšší prioritou je vždy před prvkem s nižší prioritou. Pokud přidáváme do kalendáře prvek, který nese stejný aktivační čas a zároveň stejnou prioritu jako některý prvek, již vložený v kalendáři, nový prvek se zařadí za tento prvek.

Inicializaci simulace a nastavení simulačních časů má na starost funkce `Init()`.

Páteřní funkcí celé simulace a knihovny je funkce `Run()`. Tato funkce spustí vlastní simulaci a dále postupuje podle následujícího scénáře [IMS, slide 176]:

- pokud není kalendář prázdný, vybere se první prvek v pořadí. Jinak simulace končí.
- aktualizuje se simulační čas na aktivační čas vybraného prvku,
- pokud je tento čas vyšší, než čas konce simulace, simulace se ukončí,
- pokud je stav procesu odpovídající nynějšímu záznamu v kalendáři ve stavu probouzení (`TASK_WAKING`), provede se vytvoření nového vlákna. Toto vlákno započne provádět danou událost, kalendář se mezitím uspí.
- pokud je proces ve stavu zastavný (`TASK_STOPPED`), nové vlákno se nevytvoří, pouze se pošle signál a proces pokračuje ve svém provádění v místě, kde skončil. Kalendář se opět uspí.
- spící kalendář čeká na signál od právě se provádějícího procesu, jakmile jej obdrží, probudí se a začne opět bežet
- kalendář vybere následující prvek v pořadí a provádění se opakuje.

4.4 Zařízení

Zařízení (*Facility*) je objekt, který poskytuje výlučný přístup, tzn. v daném okamžiku může zařízení vlastnit maximálně jeden proces [IMS, slide 183]. V naší knihovně obsahuje každé zařízení svoji prioritní frontu, kde se hromadí procesy, které mají zájem o zařízení. Jsou seřazeny tak, že více prioritní procesy obsadí zařízení dříve než ty méně prioritní. Každé zařízení má dva možné stavy: obsazené a volné.

Zařízení je možné definovat:

```
struct facility_t fac;
```

Základní operací se zařízením je jeho obsazení. Tuto činnost provádí funkce `Seize()`. Pokud je zařízení volné, obsadí se, není-li volné, proces se zařadí do vstupní fronty zařízení a uspí se. Kalendáři je poslán signál, aby mohl zpracovat další prvek.

Další důležitou operací je funkce `Release()`, která provede uvolnění zařízení. Toto nicméně může provést jen proces, který zařízení obsadil. Při uvolňování se, pokud není vstupní fronta zařízení prázdná, provede obsazení zařízení procesem, který je ve frontě na řadě. Tento prvek se také naplánuje do kalendáře na aktuální simulační čas.

4.5 Sklad

Sklad (*Store*) umožňuje simultánní přístup ke zdroji s určitou kapacitou [IMS, slide 187]. Je možné obsadit sklad více procesy, pokud na to dostačuje jeho kapacita. Pokud proces žádá o méně jednotek, než je aktuální kapacita skladu, ihned může požadovanou část obsadit. Pokud žádá vyšší počet jednotek nežli je aktuální kapacita, musí počkat, až obsazené jednotky vrátí jiné procesy.

Sklad je možné definovat:

```
struct store_t store;
```

Procesy nakládají se skladem dvěma funkcemi. Funkce `Enter()` se pokusí zabrat daný počet jednotek skladu, pokud se to nepodaří, proces se uspí, do té doby, než dostane signál, že je žádaná kapacita taková, že si může zabrat požadovaný počet jednotek.

Funkce `Leave()` uvolňuje zadanou kapacitu. Pokud zjistí, že fronta skladu není prázdná, vybere první prvek, který může sklad obsadit.

4.6 Statistiky

Zařízení i sklad si uchovávají statistiky, které je pak možné použít například k měření vytíženosti. Statistiky se v podstatě skládájí z pole, kde jsou uloženy časy, jak dlouho proces čekal na obsazení zařízení/skladu. Tyto hodnoty je možné použít k dalším výpočtům.

Z naměřených hodnot je možné v naší knihovně určit:

- součet
- průměr
- maximální hodnota
- minimální hodnota
- standardní odchylka

Navíc je možná zobrazit histogram, který naměřené hodnoty znázorní graficky.

Statistiky a histogram je možné volitelně tisknout s hlavičkou nebo bez ní. Dalším pohodlným nastavením je možnost nastavit výstup do zadaného souboru. Za tímto účelem je možné použít funkci `output_file()`. Jejím parametrem je pak název vytvářeného souboru.

Kromě práce se statistikami obsahuje tato část programu také funkce, které generují náhodná čísla. V knihovně jsou k dispozici následující funkce:

- `Random()`, generující pseudonáhodné číslo s rovnoměrným rozložením z intervalu $< 0; 1)$
- `Uniform()`, generující pseudonáhodné číslo s rovnoměrným rozložením z intervalu $< M; N)$
- `Normal()`, generující pseudonáhodné číslo s normálním rozložením
- `Exponential()`, generující pseudonáhodné číslo s exponenciálním rozložením

5 Podstata simulačních experimentů a jejich průběh

Pro testování funkčnosti knihovny a jejích funkcí bylo vytvořeno několik jednoduchých modelů SHO. Tyto modely využívaly `Facility` (zařízení) nebo `Store` (sklad), které byly obsazovány procesy. Jelikož je naše knihovna omezena operačním systémem (není možné vytvořit neomezený počet vláken, resp. procesů), testovací programy mají pevný počet procesů v systému. Testovací programy mají velmi jednoduché chování, kdy procesy obsazují zařízení (resp. sklad), čekají a poté uvolní zařízení (resp. sklad). Cílem knihovny ani testování nebylo vytvořit

model, jehož chování by bylo zkoumáno, nýbrž na jednoduchém modelu ověřit, zda implementace není v rozporu s principy diskrétní simulace.

5.1 Testování zařízení `Facility`

Pro testování správnosti implementace zařízení byl vytvořen SHO, ve kterém existuje jediné zařízení s prioritní frontou, do které jsou řazeny procesy, které nemohou být obslouženy. Vzhledem k systémových omezením (viz výše) byl v systému pevný počet procesů. V testovacím programu je vytvořeno 10 procesů, kvůli možnosti testování i na referenčním stroji `merlin.fit.vutbr.cz`.

5.1.1 Testovaný program

```
/* chovani procesu */
void * Behavior(void *)
{
        Seize(&fac, CURRENT());  /* obsazeni zarizeni */
        Wait(Exponential(1.25)); /* cekani s~exponencialnim rozlozenim */
        Release(&fac);           /* uvolneni zarizeni */
}

int main(void)
{
        /* inicializace zarizeni */
        /* inicializace simulace */
        /* generovani procesu s~prioritou */
        /* beh simulace */
        /* tisk statistik */
        /* zruseni zarizeni */
}
```

5.1.2 Výsledek simulace

```
Stats for Facility
Statistics
----------
Sum:                 54.78 min
Average:              5.48 min
Maximum:              8.73 min
Minimum:              0.00 min
Standard deviation:   2.34 min

Histogram
---------
<   0.00;   2.18 ): *
<   2.18;   4.36 ): **
<   4.36;   6.55 ): ****
<   6.55;   8.73 ): **
<   8.73;  10.91 ): *
```

5.2 Testování skladu `Store`

Pro testování správnosti implementace skladu byl opět vytvořen SHO, ve kterém existuje pouze jediný sklad s prioritní frontou, do které jsou řazeny procesy, které nemohou být obslouženy. Z této fronty jsou procesy vybírány tak, že první uspokojitelný proces je obsloužen.

5.2.1 Testovaný program

```
/* chovani procesu */
void * Behavior(void *)
{
        Enter(&store, CURRENT()); /* obsazeni kapacity */
        Wait(Exponential(1.25));  /* cekani s~exponencialnim rozlozenim */
        Leave(&store);            /* uvolneni kapacity */
}

int main(void)
{
        /* inicializace skladu */
        /* inicializace simulace */
        /* generovani procesu s~prioritou */
        /* beh simulace */
        /* tisk statistik */
        /* zruseni skladu */
}
```

5.2.2 Výsledek simulace

```
Stats for Store
Statistics
----------
Sum:                156.80 min
Average:             15.68 min
Maximum:             38.93 min
Minimum:              0.00 min
Standard deviation:  13.05 min

Histogram
---------
<   0.00;   9.73 ): *****
<   9.73;  19.46 ): **
<  19.46;  29.20 ):
<  29.20;  38.93 ): **
<  38.93;  48.66 ): *
```

5.3 Závěry experimentů

Byly opakovaně prováděny experimenty s vytvořenými jednoduchými SHO za účelem testovat správnost implementace knihovny. Na základě porovnání dosažených výsledků s *ruční* simulací lze říci, že experimenty využívající implementovanou knihovnu vykazující stejné výsledky (samozřejmě lišící se náhodnými hodnotami). Experimenty s každým modelem byly prováděny opakovaně (kvůli vyloučení možnosti, že dosažený výsledek byl jedenkrát náhodou správný, ale jindy by byl vždy chybný) a simulace vykazovaly stejné výsledky. Lze z toho usoudit, že implementované zařízení (resp. sklad) a procesy mají takové chování, jak je uvedeno v [IMS, slide 124 - 208].

6 Shrnutí simulačních experimentů a závěr

Byla vytvořena knihovna pro diskrétní simulace založené na přepínání procesů. Tato knihovna uživateli nabízí funkce, pomocí kterých si může vystavět libovolný systém hromadné obsluhy sestávající se zařízení, skladů (ty mají svoje fronty) a procesů. Knihovna si nekladla za cíl napodobit SIMLIB/C++ [4], jak v implementaci, tak ve všech schopnostech. Implementovaná knihovna zahrnuje pouze diskrétní simulace, kdežto SIMLIB/C++ nabízí i spojité a další simulace. Přesto však pro uživatelskou přívětivost byly názvy funkcí voleny tak, aby se shodovaly s funkcemi ze SIMLIB/C++.

Vzhledem k omezením, která operační systém klade bežící simulaci, není možné provádět simulace s velkým množstvím procesů (počet procesů je přímo-úměrný dostupným vláknům v systému). Experimenty s knihovnou byly proto vždy prováděny s takovým počtem procesů, pro které bylo možné vytvořit nová vlákna.

Dokumentace ke knihovně pro diskrétní simulace obsahuje popis jednotlivých funkcí, které může uživatel využívat, včetně popisu principů funkce simulace a jejích prvků. Knihovna obsahuje mnohé další interní funkce, které však pro uživatele nejsou podstatné (nicméně jsou nezbytné pro funkci knihovny); tyto funkce nejsou v dokumentaci uváděny.

Při provádění testování knihovny byly vytvořeny jednoduché modely SHO, které využívaly funkce knihovny. Během testování knihovny nebyl kladen důraz na rychlost simulace, ale na správnost postupu simulace. Také cílem testování knihovny nebylo srovnání se knihovnou SIMLIB/C++. Testovací programy, které byly za použití knihovních funkcí vytvořeny, tak ověřily, že knihovna provádí činnost simulace správně podle předpokladů.

Reference

[1] W. Richard Stevens, *Advanced Programming in the UNIX Environment*. Addison Wesley, Massachusetts, 2nd Edition, 2008.

[2] Zdeňka Rábová, et al. *Modelování a simulace*. Nakladatelství Vysokého učení technického v Brně, Brno, 3. přepracované vydání, 1992.

[3] man 7 pthread, `<http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html>` [online].

[4] SIMLIB/C++, `<http://www.fit.vutbr.cz/ peringer/SIMLIB/>` [online].