Implementace a testování linuxového plánovače úloh

Kamil Kolakowski

Listopad 2014

Úvod

Tento text vzniknul jako studijní materiál pro techniky testující plánovač úloh. Důraz je zde kladen na rozšíření povědomí o implementaci současného plánovače úloh. Součástí textu je vysvětlení co na současném plánovači úloh testujeme a jakých výsledků dosahujeme.

Obsah

Seznam obrázků

1 Linuxové jádro, úloha plánovače, druhy úloh

1.1 Linuxové jádro a plánovač úloh

Současné linuxové jádro je více úlohové. Jelikož je úloh na víceúlohových systémech často více než množství procesorů v systému, musí se úlohy o čas na procesoru dělit. Plánovač úloh je algoritmus jádra systému, který se stará o co nejefektivněji využití procesorového času.

Ačkoli byl Linux původně vyvíjen jako desktopový operační systém, dnes jej můžeme najít na serverech, vestavěných zařízeních, mainframech a superpočítačích. Zatížení úlohami se na jednotlivých platformách velice liší. Z tohoto důvodu a z důvodu prosazování se nových technologií jako je například multiprocessing, symetrický multithreading, nerovnoměrný přístup do paměti (numa), dochází k přehodnocování strategie přidělování zdrojů procesům.

Každý plánovač také řeší rovnováhu mezi uživatelskou odezvou a obecnou férovostí. Po zamyšlení se nad výše jmenovanými vlastnosti a technologiemi je snadnější pochopit, jak složitý může být problém plánování úloh.

1.2 Procesy a vlákna

Procesy jsou v Linuxu v rámci plánování brány jako skupiny vláken, které sdílí skupinu vláken (TGID). Jádro systému plánuje vždy zvlášť jednotlivé vlákno a ne proces. V celém mém dokumentu budu používat pojem úloha namísto vlákna nebo procesu.

1.3 Členění úloh

Úlohy si rozdělíme dle zdrojů, které požadují a podle toho jakým způsobem musí být plánovány, jelikož se v celém následujícím textu budeme věnovat jen jednomu typu úloh.

1.3.1 Normální a úlohy reálného času

Systém plánující úlohy reálného času (real time system) je takový systém, který garantuje doby odezvy systému na události, což znamená, že každá oparace by měla být dokončena v daném časovém období. Systém je klasifikován jako hard real time, pokud zmeškání lhůty může způsobit selhání systému a systém nazýváme soft real time v případě, že systém může tolerovat několik zmeškaných časových omezení. V běžném linuxovém kernelu je podpora pouze pro soft real time úlohy (systém je soft real time).

Real time úlohy mají v linuxovém plánovači svoji vlastní třídu pravidel (`kernel/sched/rt.c`).

Ačkoli současný linuxový plánovač obsahuje i pravidla pro plánování úloh reálného času, nebudu se jim v této práci věnovat. Mnohem důležitější a v běžném praxi více používané jsou úlohy normální v rámci terminologie linuxového plánovače others úlohy.

Normální úlohy, které známe z desktopů, žádné časové omezení nemají.

1.3.2 Úlohy CPU a I/O závislé

Z hlediska plánování procesů rozdělujeme na úlohy závislé na vstupně výstupních zařízeních (IO závislé) a na úlohy závislé na CPU. Některé úlohy tráví většinu času na CPU, na kterém dělají výpočty, jiné úlohy tráví hodně času tím, že čekají než se dokončí vstupní či výstupní operace, které jsou obvykle velmi pomalé. Operace I/O mohou čekat na vstup z klávesnice, disku nebo sítě. Záleží na tom na co je systém určen. Na desktopových systémech se často setkáváme s úlohami, které jsou I/O závislé na výpočetních serverech jsou to nejčastěji úlohy CPU závislé.

Jelikož je Linux navržen tak, aby běžel na různých systémech, musí se umět vypořádat se všemi těmito úlohami. Úlohy, které běží dlouhou dobu, dokončí více práce na úkor odezvy systému. Jestliže naopak úloha běží krátkou dobu, systém může odpovědět rychleji na I/O požadavek, propustnost systému je nižší, protože v systému rostou režie plánovače na výměnu úloh. Plánovač musí proto hledat rovnováhu mezi odezvou a celkovou propustností systému.

V této práci se zaměříme pouze na úlohy závisle na CPU.

2 Předchůdci CFS

- První plánovač (jádro verze 1.2, rok 1995) byl založený na round robin plánovací technice. Z jeho minimalistického návrhu plynula jednoduchost a rychlost, ale nebylo to komplexní řešení plánování, nebyl zaměřen na architektury s mnoha procesory ani na architektury s podporou více vláken.

- U plánovače úloh od verze jádra 2.2 (rok 1999) došlo k implementaci tříd plánování, umožňující používat rozdílná pravidla pro real-time úlohy, non-preemtible úlohy a non-realtime úlohy.

- Jádro 2.4 (rok 2001) má jednoduchý plánovač operující s O(N) časem. Plánovač rozdělil čas do epoch a v rámci této epochy bylo úloze povoleno běžet její časové kvantum. Jestliže úloha nepoužila celé své časové kvantum, pak polovina zbývajícího času byla přičtena do k novému časovému kvanta, aby mohla úloha běžet déle v další časové epoše. Plánovač přes všechny úlohy aplikoval funkci goodness k zjištění jakou úlohu má spustit jako další. Ačkoli toho lze dosáhnout snadno, byl příliš neefektivní, postrádal škálovatelnost a byl špatný na úlohy reálného času. Jeho další slabinou bylo postrádání podpory pro nové hardwarové architektury jako jsou třeba více jádrové procesory.

- Prvním plánovačem na jádrech verze 2.6 (rok 2003) byl plánovač zvaný O(1), který byl navržen k řešení mnoha problémů 2.4 plánovače. Nejdůležitější změnou bylo, že plánovač již nezahrnoval identifikaci další úlohy pro spuštění (a nutnost opakovaně procházet všechny úlohy, aby zjistil, kterou má spustit nejdřív), z čehož plynulo i jeho jméno O(1), což znamená, že pracoval v konstatním čase a byl více škálovatelný. O(1) plánovač uchovával záznamy o běžících úlohách ve frontách běžících úloh (front bylo více pro každou úroveň priorit dvě – jedna pro aktivní úlohy, druhou pro expirované úlohy), které používal pro výběr, jakou úlohu bude provádět jako další. Plánovač také zahrnoval měření interaktivity s množstvím heuristiky, která způsobila, že byl plánovač těžkopádný. Plánovač zvýhodňoval interaktivní úlohy před dávkovými (batch) úlohami.

3 Completely fair scheduler - úplně férový plánovač

Byl poprvé použit v jádrech 2.6.23 v roce 2008. Autorem je Ingo Molnár autor O(1) plánovače.

3.1 Kostra plánovače

Vstupní branou do linuxového plánovače úloh je funkce `schedule()`, která je definovaná v `kernel/sched/core.c` (v dřívějších kernelech v `/kernel/sched.c`). Tuto funkci používá jádro k vyvolání procesu plánovače, který rozhoduje o tom, která úloha se má spustit a poté realizuje její spuštění.

Jelikož je linuxové jádro pre-emptivní, může se stát, že je úloha přerušena v režimu jádra úlohou s vyšší prioritou. Zastavená úloha v nedokončené operaci v režimu jádra může pokračovat, až když je znova naplánována.

Proto první věc kterou funkce `schedule()` udělá je, že vypne pre-emptci, takže úloze nemůže být odebráno CPU během kritických operací.

```
preempt_disable();
```

Dále je pak uzamčena fronta úloh na aktuálním CPU, jelikož pouze jedné úloze je povoleno modifikovat frontu úloh.

```
raw_spin_lock_irq(&rq->lock);
```

Potom funkce schedule() zjišťuje stav úlohy, která běžela jako předchozí. Jestliže současná úloha již nechce běžet a zároveň je povolena preemce, pak může být úloha odebrána z běžící fronty.

```
if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
```

Jestliže úloha přijala neblokovaný signál, pak je její stav nastaven na běžící a úloha je ponechána ve frontě běžících úloh. Takže voláním prev má šanci být naplánována znova.

```
if (unlikely(signal_pending_state(prev->state, prev))) {
   prev->state = TASK_RUNNING;
```

K odebrání úlohy z fronty úloh voláme funkci `deactivate_task`, která interně volá `dequeue_task()` pro danou plánovací třídu.

```
deactivate_task(rq, prev, DEQUEUE_SLEEP);
```

Dále voláme `pick_next_task` k výběru další vhodné úlohy, která bude spuštěna na CPU.

```
next = pick_next_task(rq, prev);
```

Poté dojde k vyčištění dvou příznaků (proměnných) sloužících k vyvolání funkce `schedule()`. Tyto příznaky jsou součástí `task_struct` a jsou pravidelně kernelem kontrolovány.

```
clear_tsk_need_resched(prev);
clear_preempt_need_resched();
```

Funkce `pick_next_task` je také implementována v `kernel/sched/core.c`. Prochází plánovací třídy a hledá třídu s největší prioritou, která má spustitelnou úlohu.

```
for_each_class(class) {
   p = class->pick_next_task(rq, prev);
```

Jelikož je většina úloh obsluhována pomocí třídy `fair_sched_class`, je zkratka do této třídy implementována na hned začátku funkce `pick_next_task()`.

```
if (likely(prev->sched_class == class &&
   rq->nr_running == rq->cfs.h_nr_running)) {
```

Takže schedule() zkontroluje jestli `pick_next_task()` našla novou úlohu nebo vzala úlohu, která již běžela předtím.

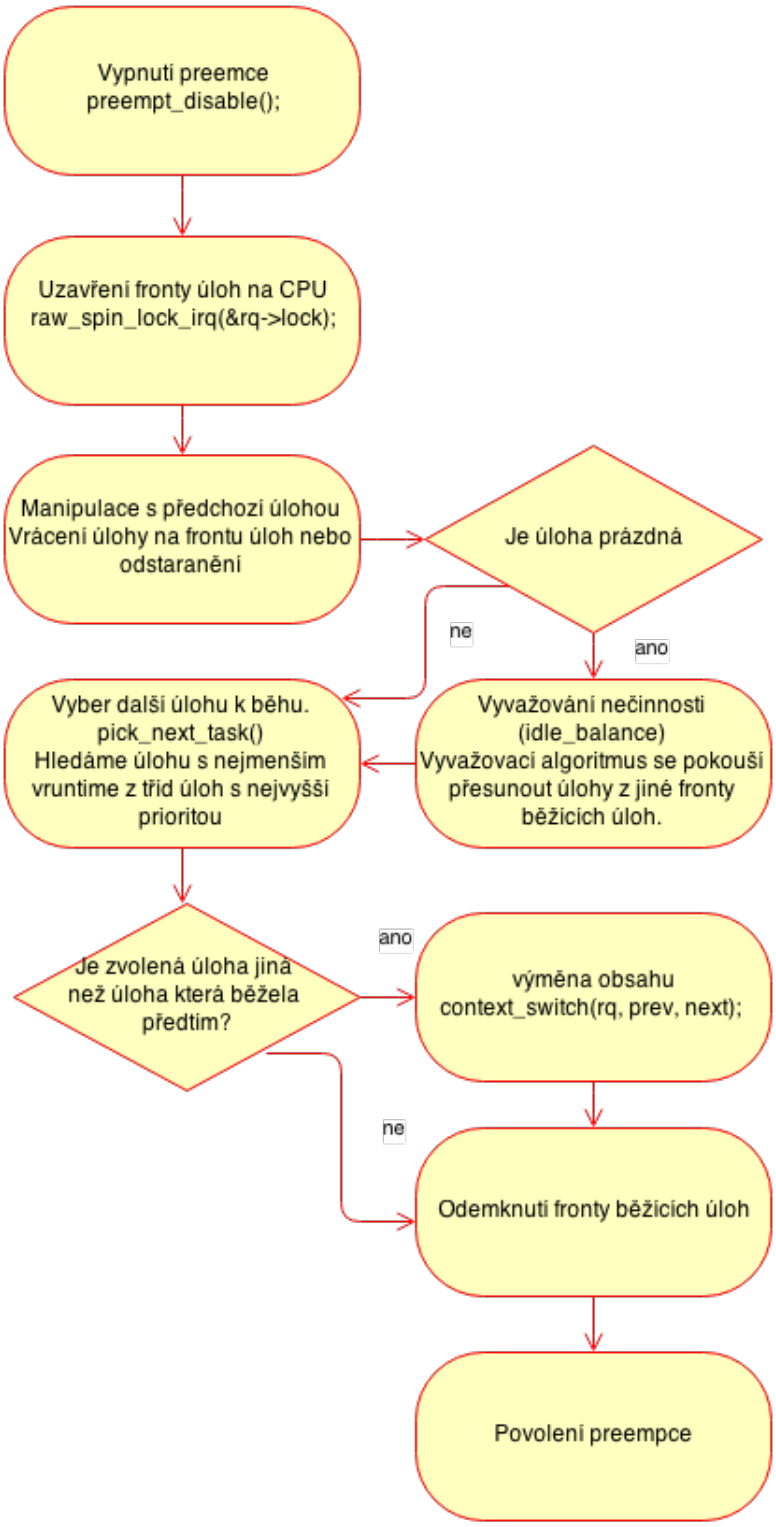
```
if (likely(prev != next)) {
```

Jestliže našla úlohu, která běžela předtím nechá ji běžet dál. Jestliže plánovač vybere úlohu, která předtím neběžela, je volána volána funkce s samovysvětlujícím názvem `context_switch`. V rámci context switche dochází k výměně stavu registrů, zásobníku a je přemapována paměť.

```
context_switch(rq, prev, next); /* unlocks the rq */
```

3.2 Plánovací třídy

Každá úloha patří do určité třídy plánování, která určuje, jakým způsobem bude úloha naplánována. Současný plánovač byl navržen tak, aby bylo možno používat rozšířenou hierarchii modulů. Jednotlivé moduly jsou zapouzdřením plánovacích pravidel, které používá kostra plánovače. Plánování třída definuje sadu funkcí prostřednictvím `sched_class` (v `kernel/sched/sched.h`), které definují chování plánovače.



Obrázek 1: Diagram algoritmu kostry plánovače

Každá třída plánování poskytuje funkce jako:

- `enqueue_task` (...) Je volána, když se úloha stává spustitelnou. Vkládá plánovací entitu (úlohu) do fronty běžících úloh (ve fair třídě do červeno-černého stromu) a zvýší hodnotu proměnnou `nr_running` o 1.
- `dequeue_task` (...) Pokud úloha není dále spustitelná, je volána tato funkce, aby vyjmula danou plánovací entitu (úlohu) z fronty běžících úloh (v fair plánovací třídě z červeno-černého stromu). Sníží se hodnota proměnné `nr_running` o 1.
- `yield_task` (...) Tato funkce je v podstatě stejná jako předchozí. Vyjme úlohu a následně ji zařadí, není-li `compat_yield` sysctl zapnutý. V tomto případě umístí plánovací entitu (úlohu) do fronty běžících úloh (ve fair třídě nejvíce napravo do červeno-černém stromu).
- `check_preempt_curr` (...) Tato funkce kontroluje, zda úkol, který se dostal do spustitelného stavu, může odebrat současně běžící úlohu.
- `pick_next_task` (...) Tato funkce vybírá nejvhodnější úkolu způsobilou ke spuštění jako další.
- `set_curr_task` (...) Tato funkce je volána, když změní-li se úloze plánovací třída nebo změní-li se skupina úlohy.
- `task_tick` (...) Tato funkce je většinou volána z `time_tick()` funkce a může vést k výměně úloh. Řídí běh preempce.

Všechny existující plánovací třídy jsou v linuxovém jádru seřazeny v seznamu podle priorit plánovacích tříd. První proměnná ve struktuře `sched_task` se nazývá `next` a je ukazatelem na plánovací třídu s nižší prioritou. Seznam je užíván k preferenci úloh jednoho typu před druhým. V současnosti vypadá seznam tříd následovně

$$stop\_class \rightarrow dl\_class \rightarrow rt\_class \rightarrow fair\_class \rightarrow idle\_class \rightarrow NULL$$

`Stop` a `idle` jsou speciální třídy plánování. `Stop` slouží k naplánování úlohy pro zastavení úlohy běžící na CPU, který odebírá všechno a nemůže být odebrán ničím. `idle` slouží k naplánování nečinné úlohy na CPU, která je naplánovaná pouze v případě, že neexistuje jiná úloha k běhu. Třída `dl(deadline)` je pro plánování úloh s určeným termínem ukončení, třída `rt(real-time)` je pro plánování úloh reálného času a `fair` slouží pro plánování běžných úloh.

Úloze lze specifikovat třídu plánování pomocí systémové utility `chrt` nebo pomocí api, které je popsáno podrobně na manuálových stránkách `man sched`.

3.3 Volání plánovače

K volání plánovače úloh dochází prostřednictvím funkce schedule() z těchto tří důvodů:

3.3.1 Pravidelné volání k aktualizaci vruntime u právě běžící úlohy

Časovač přerušení pravidelně volá fuknci `scheduler_tick()` (z `kernel/sched/core.c`). Dochází k aktualizaci hodnoty vruntime a také ke změnám na aktuální frontě úloh. V kódu funkce `scheduler_tick()` je aktualizace realizována voláním

```
curr->sched_class->task_tick(rq, curr, 0);
```

Takto volaná aktualizace nám zaručuje aktualizaci údajů dle pravidel dané třídy, ve které byla úloha spuštěna. Ve funkci `scheduler_tick()` jsou volání

```
rq->idle_balance = idle_cpu(cpu);
trigger_load_balance(rq);
```

která zajištují vyvážení zátěže v případě, že máme je systém s více CPU.

3.3.2 Když současně běžící úloha usne

Úloha která je uspávána, čeká na specifickou událost, je implementována podobně jako následující ukázka, kterou lze najít v různých částech linuxového jádra.

```
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) /* waiting for event */
{
     prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
     schedule();
}
finish_wait(&q, &wait);
```

Úloha vytvoří čekací frontu a přidá se do ní.

```
DEFINE_WAIT(wait);
add_wait_queue(q, &wait);
```

Ve smyčce pak čeká, než je splněna podmínka. Jestliže nenastala potřebná událost, zavolá se plánovač a úloha se uspí. Ve smyčce dále dojde k nastavení statusu úlohy na TASK_INTERRUPTIBLE nebo TASK_UNITERRUPTIBLE.

```
while (!condition) /* waiting for event */
{
     prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
     schedule();
}
```

Plánovač odstraní úlohu z běžící fronty. Nastane-li potřebná událost (podmínka condition), je ukončena smyčka a úloha je odstraněna z čekací fronty. Tyto činnosti jsou prováděny ve funkci `schedule()`. Úloha čeká na jinou úlohu, která zavolá funkci `wake_up(&wait)`

3.3.3 Při probuzení spící úlohy

Spící úloha obvykle očekává kód, který volá funkci `wake_up` na patřičné čekací frontě. Postupně je volána hierarchie funkcí, až dojde na volání `ttwu_queue` z `try_to_wake_up()`. Ta obstará samotné probuzení úlohy.

Postupně dochází k následujícím činnostem

1. V případě, že úloha ještě není ve frontě běžících úloh, dojde k vrácení úlohy do této fronty. Toto je realizováno pomocí funkce `tttwu_queue`, která zavře frontu běžících úloh a zavolá `ttwu_do_activate()`.

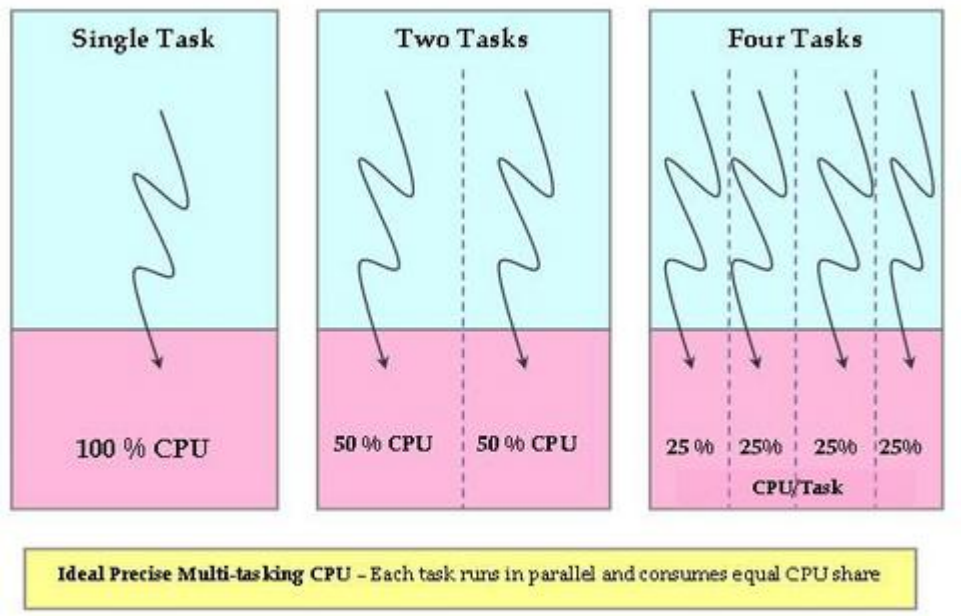
2. Probudí úlohu pomocí nastavení stavu na TASK_RUNNING (ve funkci `ttwu_do_wakeup()`).

3. Jestliže má probuzená úloha větší prioritu než úloha současně běžící, je nastaven `need_resched` příznak pro vyvolání funkce `schedule()`. Toto je také realizováno z funkce `ttwu_do_wakeup()`.

3.4 Fair plánovací třída

Zcela férový plánovač, je nejčastěji používaný pro kategorii normálních (v terminologii CFS others) úloh. Užívá plánovací pravidla z třídy fair (kterou lze najít v `kernel/sched/fair.c`). CFS je postaven na myšlenkách ideálního

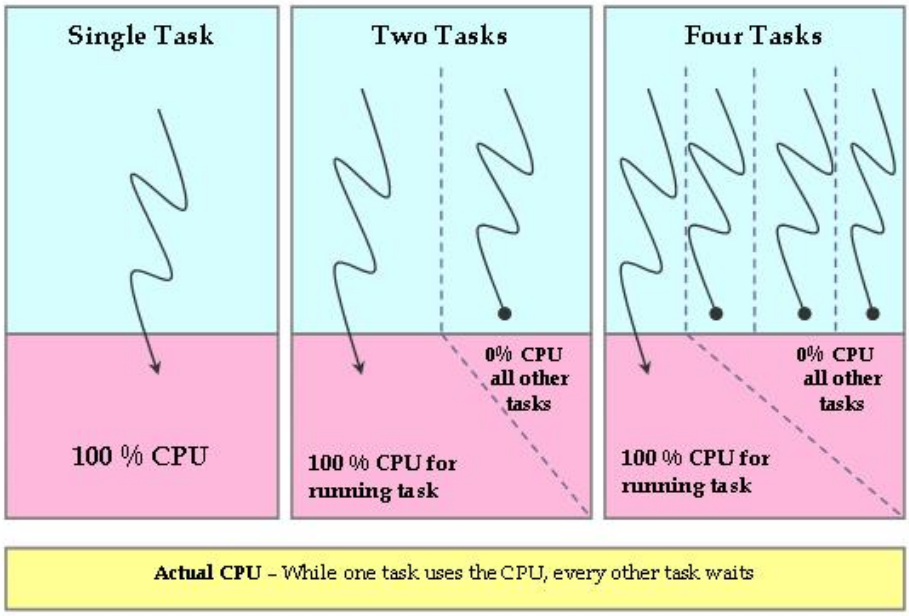
mnoho úlohového procesoru. Ideální mnoho úlohový procesor je pak takový, na kterém běží paralelně všechny aktivní úlohy a každá úloha tak dostává adekvátní porci procesorového výkonu. Takže například pro 2 úlohy běžící na jednom ideálním mnoho úlohovém systému by každá úloha dostala 50% výkonu procesoru.



Obrázek 2: Schéma ideálního procesoru

Mnoho úlohový ideální systém, je čistě abstraktní model a není z fyzikálního hlediska možný. Ale koncept CFS je založen na stejném cíli a to přidělení všem běžícím úlohám férové množství procesoru. Současný plánovač tudíž nenabízí každé úloze stejnou část výkonu procesoru, ale snaží se docílit, aby úlohy běžely na CPU stejné časové kvantum.

V CFS byl klasický model předchozích plánovačů založených na obdobích (epochs) a pevně daných časových úsecích (time slices) zcela předělán. Byl zaveden virtual runtime počítadlo pro každou úlohu, které nám udává hodnotu času stráveného na procesoru vynásobenou koeficientem priority. CFS odkazuje na vruntime (`p->se.vruntime` kde p je struktura `sched_entity`). Když se zjistí, že některá úloha z fronty běžících úloh strávila na procesoru méně času než ta právě běžící, dojde k naplánování běhu úlohy s menší hodnotou vruntime. Místo aby plánovač udržoval úlohy ve frontě, jako to dělali předchůdci CFS plánovače, CFS udržuje časově seřazený červeno-černý strom. Je samovyvažovací a neexistuje v něm cesta, která je více než dvojnásobně

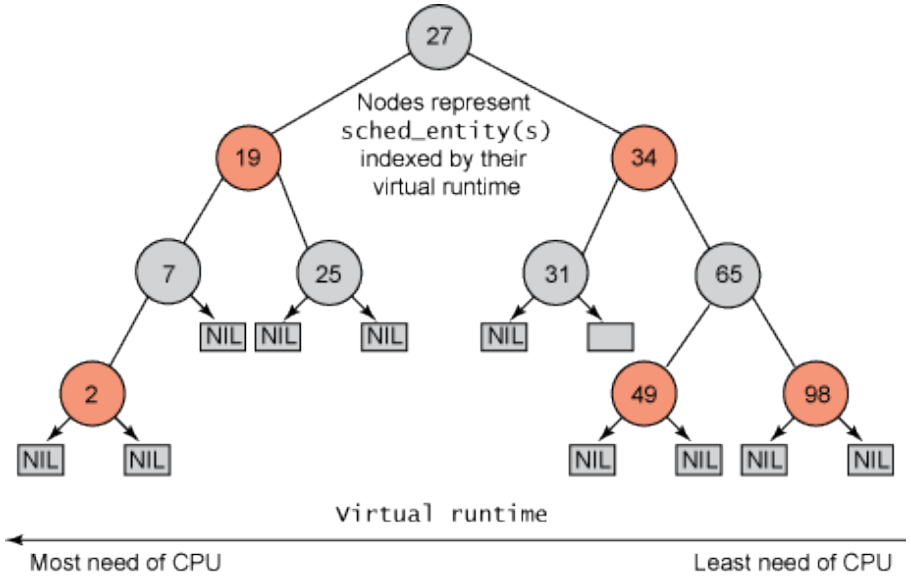


Obrázek 3: Schéma reálného procesoru

delší než kterákoli jiná v daném stromu. Druhá vlastnost je, že jakékoli operace na tomto stromu probíhají v O(log n) čase (kde n je počet uzlů v daném stromu).

Úlohy (representovanými `sched_entity` objekty) jsou na stromu seřazeny dle hodnot ve vruntime. Úlohy s nejvyšší potřebou běhu na CPU jsou uloženy na nejvíce levé části stromu a úlohy s nejnižší potřebou běhu na CPU jsou uloženy ve stromě nejvíc napravo.

Aby byl plánovač férový, vezme pro spuštění první úlohu zleva. Čas po který běžela současná úloha, vynásobený koeficientem priority, se přičte k virtuálnímu času běhu a pokud je úloha běžící je vložena zpět do červeno-černého stromu. Úlohám na levé straně stromu je dán čas na procesoru (postupně vždy po jedné úloze) a obsah stromu přebíhá zprava doleva k udržení férovosti. Jednoduše řečeno, všem úlohám je postupně přidělován čas na procesoru (dle pořadí ve stromu zleva doprava) a neustále dochází k vyvažování stromu všech běžících úloh.



Obrázek 4: RB strom úloh setřízený podle hodnot vruntime

3.5 Plánování skupin úloh

Další velice zajímavou vlastností CFS je koncept plánování skupin úloh. Poprvé implementováno v jádrech 2.6.24.

Plánování skupin je nový způsob jak obstarat férovost v plánování, když se úloha rozvětví do více úloh. Příkladem tohoto může být HTTP server, který se rozvětví na mnoho úloh za účelem obsluhovat HTTP požadavky paralelně (což je typické pro HTTP servery). Místo aby byly brány všechny úlohy jako úlohy se stejnou váhou, CFS zavádí skupiny, aby takové chování ošetřil. Proces, který větví úlohy, sdílí jejich vruntime v rámci skupiny a to hierarchicky, zatímco každá další jednotlivá úloha udržuje svůj vlastní vruntime. Tímto způsobem každá další úloha dostává přibližně stejné množství plánovaného času jako skupina. V systému můžeme najít `/proc` rozhraní, kterým řídíme hierarchii procesu a které nám dává plnou kontrolu nad tím jak jsou skupiny procesů tvořeny. Pomocí této konfigurace můžeme vytvořit férovost přes uživatele systému, přes procesy nebo jejich kombinace.

Rozdělování úloh do skupin je v současném kernelu automatické. Lze ho zapnou a vypnout v `/proc/sys/kernel/sched_autogroup_enabled`.

Obrázek 5: Příklad vytváření skupin multimedia a browser pomocí cgroups

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/cpu
# mount -t cgroup -ocpu none /sys/fs/cgroup/cpu
# cd /sys/fs/cgroup/cpu

# mkdir multimedia  # timto vytvorime skupinu uloh multimedia
# mkdir browser     # timto vytvorime skupinu uloh multimedia

# Timto zajistime, ze multimedia budou dostavat
# 2x vice procesoroveho casu nez skupina browser

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &  # Spustime firefox presuneme ho
#            # do skupiny "browser"
# echo <firefox_pid> > browser/tasks

# #spustime gmplayer
# echo <movie_player_pid> > multimedia/tasks
```

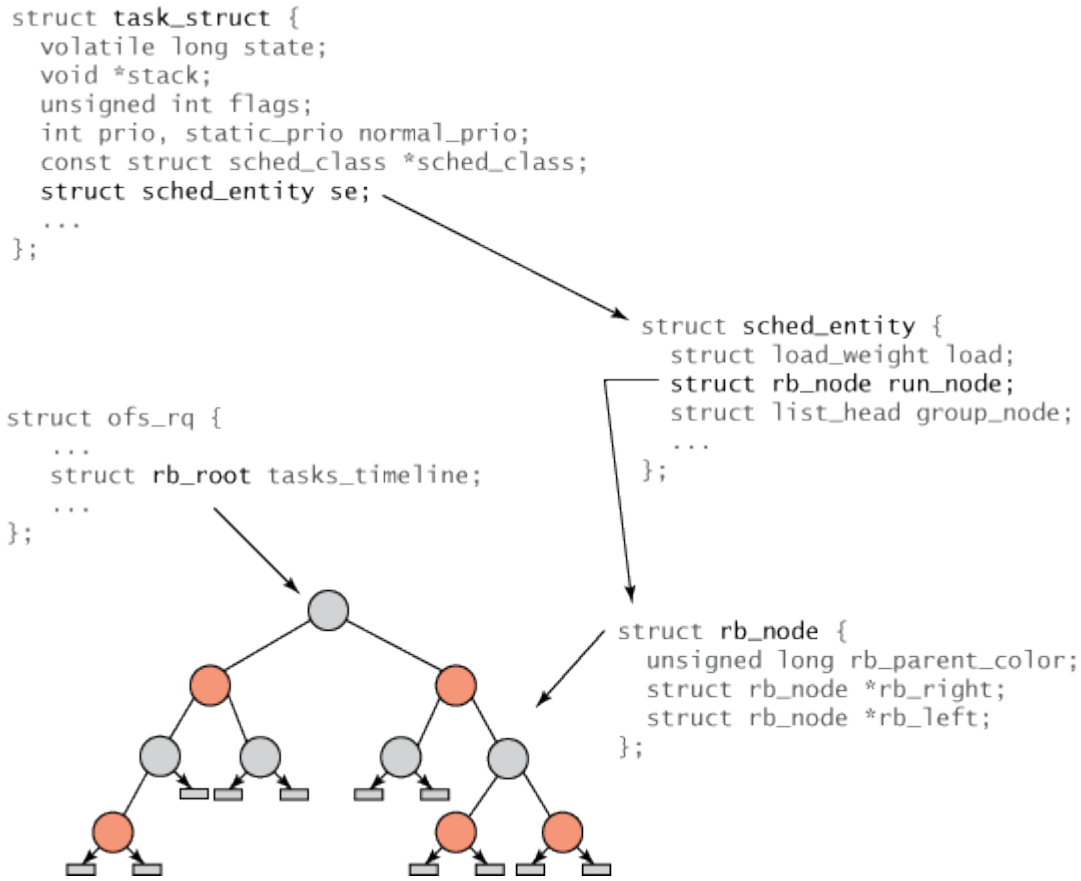
3.6 Priority úloh

CFS nepoužívá fronty úloh pro každou prioritu jak to dělali předchůdci CFS, ale priority jsou řešeny tak, že úlohám s nižší prioritou roste hodnota vruntime rychleji a naopak úlohám s prioritou vyšší roste hodnota vruntime pomaleji.

3.7 Datové struktury v CFS

Všechny úlohy jsou v Linuxu reprezentovány strukturou `task_struct`. Tato struktura (spolu s dalšími ze kterých je samotná `task_struct` složená) plně popisují úlohu a zahrnují aktuální stav úlohy, její zásobník, příznaky procesu, prioritu (statickou i dynamickou). Tyto informace a mnohem více podobných

struktur lze najít v `include/linux/sched.h`. Z důvodu aby bylo možno plánovat úlohy podle různých pravidel (plánovacích tříd), nenalezneme CFS závislé informace v `task_struct`. Místo toho byla vytvořená nová struktura pojmenovaná `sched_entity` (také z `include/linux/sched.h`) k uchovávání informací o plánování.



Obrázek 6: Schéma složení struktur plánovače

Na kořen stromu je odkazováno pomocí `rb_root` struktury (z `/include/linux/rbtree.h`) z `cfs_rq` struktury (v `kernel/sched/sched.h`). `cfs_rq` obsahuje také ukazatel na nejvíce levý prvek ve stromu `min_vruntime` (který má nejmenší hodnotu vruntime), ukazatele na předchozí a současně běžící úlohu a další informace týkající se skupin a SMP plánování a vyvažování. Samotná priorita procesu je uchována v datové struktuře `load_weight`.

Uzly RB stromu představují jednu nebo více úloh, které jsou spustitelné. Každý uzel stromu je reprezentován `rb_node` (z `/include/linux/rbtree.h`) strukturou, která neobsahuje nic víc než odkaz na potomka a barvu předka. Entita `rb_node` je součástí `sched_entity` struktury, která zahrnuje `rb_node`

odkaz a mnoho různých statistických dat. Nejdůležitější položkou `sched_entity` je vruntime (64bit políčko), která znázorňuje dobu, po kterou proces běžel, a slouží jako index pro červeno-černý strom. Struktura `task_struct` leží na vrcholu hierarchie a plně popisuje úlohu a zahrnuje `sched_entity` strukturu.

CFS dále udržuje hodnotu `rq->cfs.min_vruntime`, která je monotónně zvyšována na minimální hodnotu vruntime všech úloh ve stromu. Celkové množství vykonané práce v systému je sledováno pomocí `min_vruntime`. Tato hodnota se používá pro umístění nově aktivovaných úloh z nejvíce levé strany stromu.

Celkové množství běžících úloh ve stromu je zaznamenáno v `r->cfs.load` hodnotě, což je suma priorit všech běžících úloh.

3.8 Aktualizace doby běhu úlohy (vruntime)

Vstupní funkci k aktualizaci vruntime je `task_tick_fair()` volána z funkce `scheduler_tick` přes odkaz `task_tick()` (z `kernel/sched/core.c`).

Z funkce `task_tick_fair` je volána funkce `entity_tick()`, která dělá dvě věci:

1. Aktualizuje vruntime současně běžící úlohy. Toto je realizováno pomocí funkce `update_curr(cfs_rq)`, která počítá čas strávený úlohou od posledního naplánování (`delta_exec`) a ten je pak použit jako parametr při volání funkce `__update_curr()`, která provede samotnou aktualizaci vruntime. K rozdílu časů běhů je započítána váha priority aktuálně běžící úlohy (váha je uložená v `load_weight`) a výsledek je uložen do vruntime aktuální úlohy. Také dochází ke aktualizaci `min_runtime` hodnoty.

2. Zjišťuje zda je potřeba současně běžící úlohu odebrat. Jakmile je vruntime aktuální úlohy aktualizován, je volána funkce `check_preempt_tick()`.

   ```
   if (cfs_rq->nr_running > 1)
      check_preempt_tick(cfs_rq, curr);
   ```

   Tato funkce vezme vruntime současné úlohy a kontroluje ji proti vruntime úlohy, která je nejvíce vlevo v červeno-černým stromu, aby zjistil, zda bude nutno aktuálně běžící úlohu vyměnit.

Funkce `sched_slice()` volaná z funkce `check_preempt_tick`

```
ideal_runtime = sched_slice(cfs_rq, curr);
```

vrací ideální délku běhu aktuální úlohy v závislosti na množství běžících úloh. Jestliže je čas posledního běhu úlohy (delta) větší než tato hodnota, je nastaven na aktuálně běžící úlohu příznak `need_resched`.

```
if (delta_exec > ideal_runtime) {
   resched_curr(rq_of(cfs_rq));
```

Pokud ne, pak se čas běhu kontroluje proti hodnotě v `min_granularity`. V případě, že úkol běžel déle než `min_granularity` a celkově je na červeno-černém stromě více než jedna úloha provede se porovnání s úlohou nejvíce nalevo v červeno-černém stromu. Jestliže jsou rozdíly mezi časy běhu těchto dvou úloh kladné, pak to znamená, že současná úloha běžela déle než úloha nejvíce nalevo. Dojde také k nastavení `need_resched` příznaku, aby mohlo dojít co nejdřív k přeplánování.

```
if (delta_exec < sysctl_sched_min_granularity)
   return;

se = __pick_first_entity(cfs_rq);
delta = curr->vruntime - se->vruntime;

if (delta < 0)
   return;

if (delta > ideal_runtime)
   resched_curr(rq_of(cfs_rq));
```

V kostře plánovače jsme se dočetli, jak byly úlohy deaktivovány a vyjmuty z fronty běžících úloh, nebo aktivovaných když se probudily v `try_to_wake_up()`. Ve fair plánovací třídě jsou volány `equeue_task_fair()` a `dequeue_task_fair()` ve funkcích `enqueue_entity()` a `dequeue_entity()` pro aktualizování uzlů v červeno-černém stromu.

Funkce `schedule()` volá funkci `pick_next_task()` určité plánovací třídy s největší prioritou, která má běžící úlohy. Jestliže ve třídě nejsou úlohy, je vrácena hodnota NULL.

```
do {
        struct sched_entity *curr = cfs_rq->curr;

       if (curr && curr->on_rq)
                update_curr(cfs_rq);
        else
                curr = NULL;
```

```
        if (unlikely(check_cfs_rq_runtime(cfs_rq)))
                goto simple;

         se = pick_next_entity(cfs_rq, curr);
         cfs_rq = group_cfs_rq(se);
} while (cfs_rq);
```

Z funkce `pick_next_task()` je volána funkce `pick_next_entity()`, která odstraní běžící úlohu z červeno-černého stromu úloh, protože běžící úloha není na stromě obsažena. While smyčka je použita pro počítání férového plánování skupin.

3.9 Shrnutí

CFS je volán v době když dojde k vytvoření nové úlohy či probuzení stávající úlohy, usnutí úlohy, nebo když je vyvoláno jako obsluha časovače přerušení. Spočítá čas právě strávený úlohou na CPU vynásobí ho koeficientem priority úlohy a přičte jej do `p->se.vruntime`. Jestliže `p->se.vruntime` poroste dostatečně a současně běžící úloha již není úlohou s nejmenší hodnotou vruntime, pak je spuštěna úloha, která je nejvíce nalevo v červeno-černého stromu. Úloha která právě běžela se vrátí zpátky do stromu. Toto není úplně přesné, jelikož CFS má nastavenou hodnotu granularity, o kterou maximálně může být větší vruntime běžící úlohy než vruntime úlohy úplně nalevo ve stromu. S granuralitou se počítá proto, aby z důvodu velmi malého rozdílu v hodnotě vruntime mezi úlohami nedocházelo k výměně obsahu a s tím souvisejícím negativním jevům jako zahozením dat ve vyrovnávací paměti atd. V systému jde granularitu konfigurovat v `/proc/sys/kernel/sched_min_granularity_ns`. Zvýšení hodnoty bude vést k větší propustnosti, ale bude mít pomalejší odezvu na interaktivní procesy (vhodné pro servery). Zmenšením hodnoty v `sched_min_granularity_ns` docílíme k rychlejší odezvě a k menší propustnosti (z důvodu častější výměny obsahu).

Větší část CFS návrhu už je mimo tento jednoduchý koncept, přibyly zde rozšíření pro vyvažování úloh na numa systémech a dalších algoritmy například na rozpoznávání spících procesů.

4 Vyvažování

4.1 Vyvažování na UMA SMP systémech

UMA – uniform memory access. Znamená, že přístup jakéhokoli procesoru do jakéhokoli segmentu paměti je vždy stejně rychlý.

Vyvažování na SMP systémech bylo implementováno s cílem zlepšit výkonnost tím, že odebereme úlohy od nejvíce vytížených CPU a přesuneme je na CPU volné nebo méně vytížené. Linuxový plánovač kontroluje pravidelně, jak jsou rozmístěné úlohy na systému a když vidí, že systém není vyvážen, spustí vyvažování.

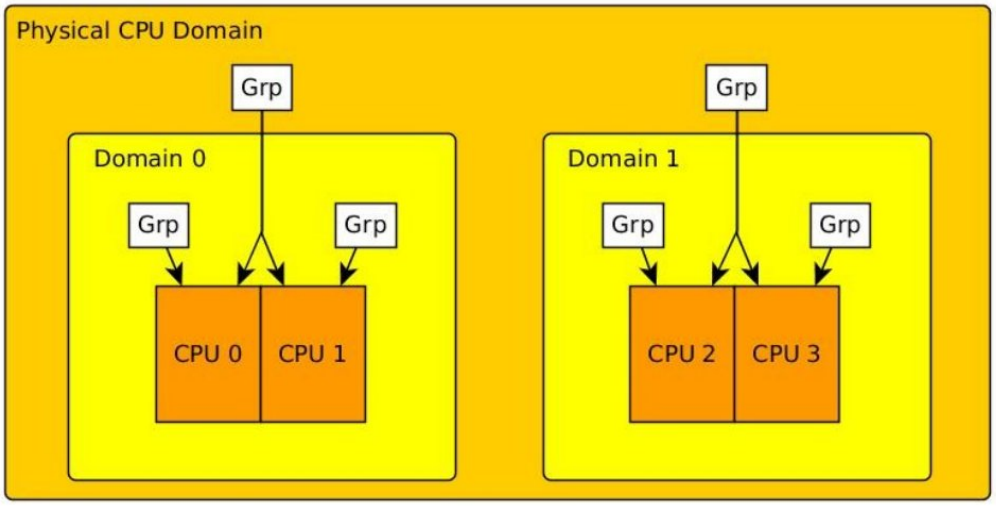
Důležitým bodem je správné chápání topologie systému plánovačem. Můžeme mít systémy s několika jádry, kde mohou úlohy trpět více vyprázdněním vyrovnávacích pamětí v důsledku přesunu úloh, než v důsledku běhu úlohy na vytíženém CPU. Jiné systémy zase mohou být více flexibilní vůči migraci úloh a to díky sdíleným vyrovnávacím pamětem (například systémy podporující hyperthreading).

Kvůli rozdílům v topologiích systémů, byly zavedeny od jádra 2.6 plánovací domény. Plánovací domény vytváří hierarchické skupiny procesorů v systému, které poskytují jádru OS přehled o topologii systému a usnadňují vyvažování.

4.1.1 Plánovací domény a skupiny

Plánovací doména je množina procesorů, které sdílejí vlastnosti a plánování pravidla, která mohou být vyvažovány proti sobě. Každá doména může obsahovat jednu nebo více plánovacích skupin, které se považují za jednotku v doméně. Takže když se plánovač snaží vyvážit zatížení v rámci domény, pokusí se vyvážit zatížení každé plánovací skupiny, bez ohledu na to co se děje ve skupině.

Představme si, že máme systém dvou fyzických procesorů, na kterých je hyperthreading, tak dostaneme celkem čtyři logické procesory. Při spuštění systému, jádro rozdělí logické jádra do doménové hierarchie druhé úrovně viz obrázek



Obrázek 7: Schéma skupin a domén

Každý hyperthreadový procesor je vložen právě do jedné skupiny a obě skupiny jsou v téže doméně. Tyto dvě úrovně domén poté tvoří celý procesor.

Diagram pro numa systémy je stejný jako předchozí diagram. Jednotlivé domény pak reprezentují numa nody.

Každá plánovací doména má vyvažovací pravidla, která jsou platná pouze na této úrovni (tzn. v této doméně).

Parametry pravidel zahrnují, jak často se máme pokusit udělat vyvážení napříč doménou, jaké je povoleno mít nevyvážené zatížení mezi skupinami procesorů, než se spustí vyvažování, jak moc je povoleno mít nevyvážené zatížení na procesoru, než se spustí vyvažování, jak dlouho může být úloha mimo procesor, než považujeme její vyrovnávací paměť za ztracenou. Existují také příznaky, které nám signalizují změny v zatížení systému.

Aktivní vyvažování zátěže je spouštěno pravidelně. Dochází k procházení hierarchie domén nahoru a zjišťuje se, zdali jsou všechny skupiny po cestě vyvážené. Když se narazí na nějakou nevyváženou, provádí vyvažování podle pravidel dané domény.

4.1.2 Implementace vyvažování

Ve `sched.h` najdeme dvě struktury, které byly přidány do kódu za účelem vyvažování. Jedná se o `sched_domain` (`include/linux/sched.h`) a pak `sched_group` (`kernel/sched/sched.h`).

```
struct sched_domain {
        /* These fields must be setup */
        struct sched_domain *parent;
        struct sched_domain *child;
        struct sched_group *groups;
        unsigned long min_interval;
        unsigned long max_interval;
        unsigned int busy_factor;
        unsigned int imbalance_pct;
        unsigned int cache_nice_tries;
        unsigned int busy_idx;
        unsigned int idle_idx;
        unsigned int newidle_idx;
        unsigned int wake_idx;
        unsigned int forkexec_idx;
        unsigned int smt_gain;

        int nohz_idle;                  /* NOHZ IDLE status */
        int flags;                      /* See SD_* */
        int level;

                ...


struct sched_group {
        struct sched_group *next;       /* Must be a circular list */
        atomic_t ref;

        unsigned int group_weight;
        struct sched_group_capacity *sgc;

        unsigned long cpumask[0];
};
```

V `include/linux/topology.h` najdeme nastavení pro příznaky a pro plánovací domény. V `sched_group` můžeme najít strukturu `sgc`, která vy-

jadřuje `sched_group_capacity`. Vyjadřuje výpočetní sílu dané skupiny. Například dva procesory budou mít hodnotu někde kolem 2, ale CPU s hyperthreadingem bude mít sílu jen kolem 1.1.

4.1.3 Aktivní vyvažování

Během aktivního vyvažování kernel prochází doménovou hierarchií. Začíná se na CPU doméně a postupuje výše. Když na určité doméně zjistí, že je nevyvážená, spustí vyvažovací operace.

Během inicializace plánovače, je vytvořena obsluha přerušení, ze které jsou volány funkce, které provádí vyvažování zátěže. Vyvažování je spuštěno ve funkci `scheduler_tick()` (`kernel/sched/core.c`) voláním `trigger_load_balance()` (`kernel/sched/fair.c`). `trigger_load_balance()` zkontroluje časovač a jestli je vyvažování potřeba vyhodí „vyjímku“ s parametrem `SCHED_SOFTIRQ`.

```
void trigger_load_balance(struct rq *rq)
{
   if (unlikely(on_null_domain(rq)))
      return;

   if (time_after_eq(jiffies, rq->next_balance))
      raise_softirq(SCHED_SOFTIRQ);
   #ifdef CONFIG_NO_HZ_COMMON
      if (nohz_kick_needed(rq))
         nohz_balancer_kick();
   #endif
}
```

Vyjímka je obsloužena funkcí `run_rebalance_domains()`, která volá `rebalance_domains()`.

```
static void run_rebalance_domains(struct softirq_action *h)
{
   struct rq *this_rq = this_rq();
   enum cpu_idle_type idle = this_rq->idle_balance ?
                                          CPU_IDLE : CPU_NOT_IDLE;

   rebalance_domains(this_rq, idle);
   nohz_idle_balance(this_rq, idle);
}
```

Funkce `rebalance_domains()` poté prochází doménovou hierarchii a volá `load_balance()`, pokud má daná doména nastavenou proměnnou `SD_LOAD_BALANCE` a vypršel již interval indikující stav vyvážení. Vyvažovací interval domény je ve vteřinách a je aktualizován po každém běhu vyvažování.

Aktivní vyvažování je o přetažení jedné nebo více úloh z přetíženého CPU na jiné méně vytížené CPU. Nedochází k výměně úloh, ale pouze k přesunu úlohy. Funkce, která provádí přesun úlohy na jiné CPU, se nazývá `load_balance()`. Jestliže najde nevyváženou skupinu, přesouvá jednu nebo více úloh na současné CPU a vrací hodnotu větší než 0.

Funkce `load_balance()` volá funkci `find_busiest_group()`, která hledá nevyváženost v dané `sched_domain` a vrací nejvíce zatíženou skupinu pokud taková existuje. Jestliže je systém vyvážený a žádná skupina nebyla nalezena funkce `load_balance()` skončí.

Jestliže funkce `find_busiest_group` vrátila nejvytíženější skupinu, pak ta je předána do funkce `find_busiest_queue()` a výstupem z funkce je fronta úloh nejvíce zatíženého logického procesoru ve skupině.

`load_balance()` poté hledá úlohu v běžící frontě úloh, kterou přesune na frontu současného procesoru voláním `move_tasks()`. Množství úloh, které mají být přesunuty jsou specifikovány jako parametr ve funkci `find_busiest_group()`. Běžně se stává, že jsou všechny úlohy „přišpendleny“ k frontě běžících úloh na daném CPU kvůli vyrovnávací paměti. V tomto případě funkce `load_balance` pokračuje v hledání, ale vynechá předchozí nalezené CPU.

Pokud je proměnná `SD_POWERSAVINGS_BALANCE` nastavena v doménových pravidlech a není-li nalezena nejrušnější skupina, `find_busiest_group()` hledá nejméně vytíženou skupinu v `sched_domain`. Procesory z této skupiny jsou poté uvedeny do klidového stavu.

4.1.4 Vyvažování nečinnosti

Vyvažování nečinnosti začíná, když se CPU stává nečinným. Je volána funkce schedule() na CPU vykonávající současnou úlohu, jestliže se jeho fronta běžících úloh vyprázdnila.

Tak jako aktivní vyvažování je i vyvažování nečinnosti `idle_balance()` implementováno v souboru fair.c. Kontroluje se zdali je průměrná doba čekání ve frontě nečinnosti větší než je cena migrace úlohy na tuto frontu. To znamená, že se kontroluje zdali má cenu vzít úlohu odjinud, nebo jestli je lepší jen počkat, jelikož je pravděpodobné, že se další úloha brzy probudí. Pokud migrace úlohy dává smysl, `idle_balance()` funguje téměř jako `rebalance_domains()`. To projde doménovou hierarchii a volá `idle_balance()` pro domény, které mají v doméně nataveny příznaky `SD_LOAD_BALANCE` a `SD_BALANCE_NEWIDLE`.

4.1.5 Výběr fronty pro novou úlohu

Dále je třeba dělat vyvažování, když se úloha probudí, nebo je vytvořená nová a potřebuje být umístěná ve frontě běžících úloh. Tato fronta musí být vybrána s ohledem na vyvážení úlohami celého systému. Každá plánovací třída implementuje svojí vlastní strategii na nakládání se svými úlohami a poskytuje funkce `select_task_rq()`, která je volána plánovačem během spuštění úlohy. Je volána ze třech různých důvodů, které jsou vyznačeny pomocí návěští odpovídající domény.

1. `SD_BALANCE_EXEC` je návěští používané funkci `sched_exec()`. Tato funkce je volána, jestliže úloha startuje jako nová systémovým voláním exec(). Nová úloha je jednoduchá pro vyvažování (neřešíme afinitu vyrovnávací paměti apod).

2. `SD_BALANCE_FORK` je návěští používané ve funkci `wake_up_new_task()`. Tato funkce je volána, když je vytvořená úloha probuzena první krát.

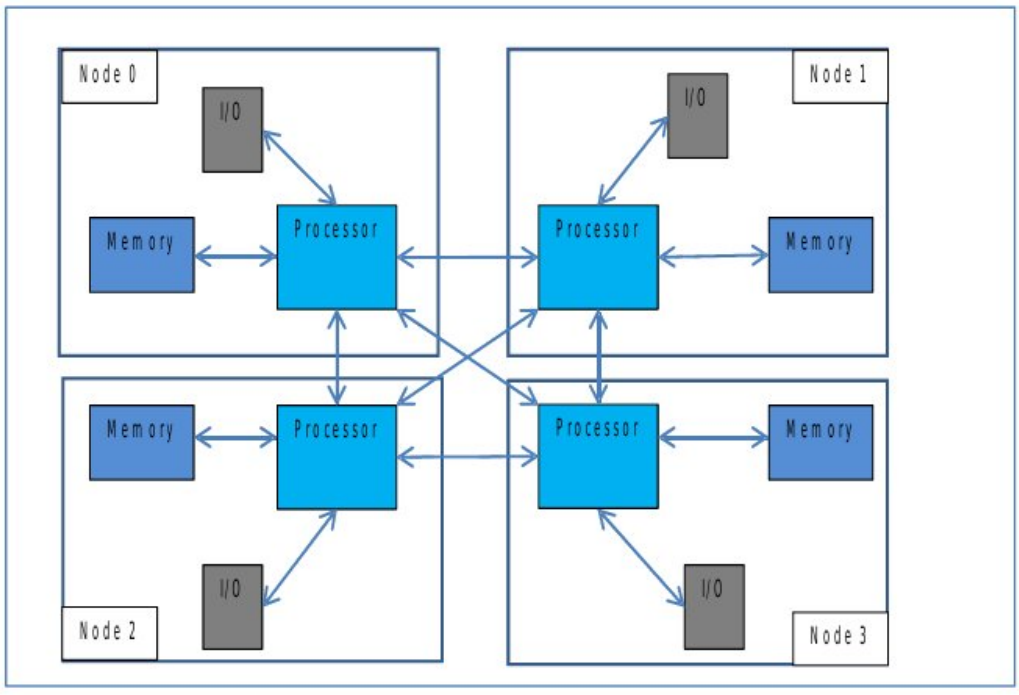
3. `SD_BALANCE_WAKE` je návěští používané ve funkci `try_to_wake_up()`. Toto je nejsložitější případ, jelikož úloha která běžela předtím, má určitou afinitu vyrovnávací paměti. Proto je třeba vybrat frontu na které budeme mít dostupnou vyrovnávací paměť.

4.2 Numa vyvažování

Non uniform memory access. Numa znamená, že přístup do různých segmentů paměti je různě rychlý. Architektura je rozšířením SMP (symetrický multi processing). Počítače začaly mít problémy s propustností mezi pamětí a procesorem v době, kdy se začaly objevovat systémy se stovkami jader, jelikož každé jádro komunikovalo stejnou sběrnicí s pamětí. Toto se podařilo vyřešit shlukováním jader a pamětí do takzvaných numa uzlů. Každý procesor (několik jader) má k dispozici svou lokální paměť, na kterou má rychlý přístup, ale má také přístup na vzdálenou paměť, kde je přístup pomalejší.

K propojování uzlů je použita sběrnice, které se obecně říká INTERCONNECT. Každý z výrobců numa systémů pojmenovává tyto sběrnice jinak INTEL používá Intel Quick Path (QPI), dříve taky používal název Common system interface (CSI), AMD používá název Hypertransport.

Výrobci systémů si sami navrhují, které numa uzly budou propojeny pomoci interconnect sběrnice na přímo, některé další jsou propojeny přes několik dalších numa uzlů.



Obrázek 8: Jednoduchá numa topologie

obrázek

Zde je výpis numactl ukazující jak máme shlukovány jádra a paměť do

numa uzlů.

```
# numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
node 0 size: 262040 MB
node 0 free: 249261 MB
node 1 cpus: 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
node 1 size: 262144 MB
node 1 free: 252060 MB
node 2 cpus: 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
node 2 size: 262144 MB
node 2 free: 250441 MB
node 3 cpus: 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
node 3 size: 262144 MB
node 3 free: 250080 MB
```

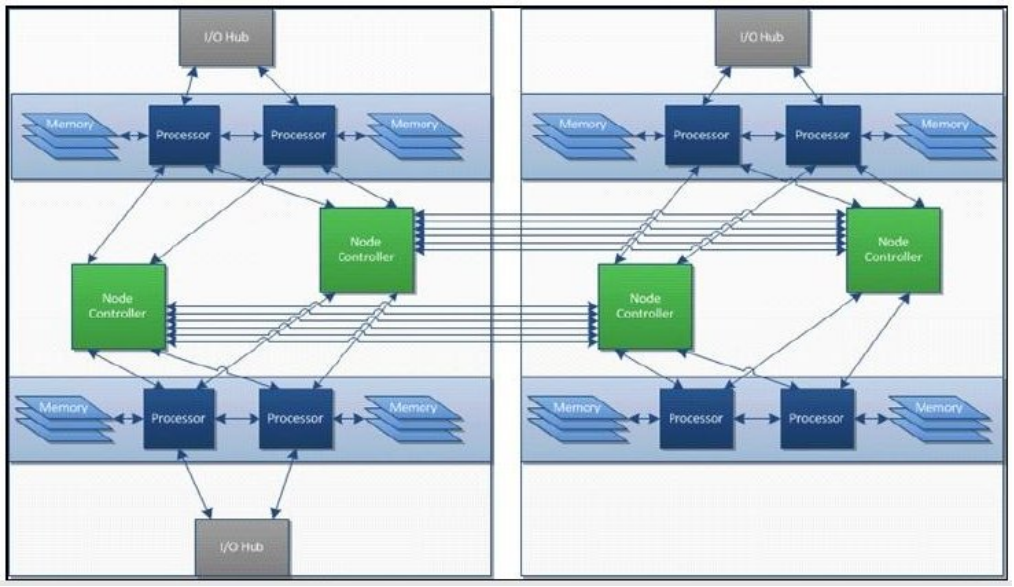
Následuje tabulka ukazující jak vzdálené (a tím i rychlé) jsou přístupy do paměti.

```
node distances:
node 0 1 2 3
0: 10 21 21 21
1: 21 10 21 21
2: 21 21 10 21
3: 21 21 21 10
```

Na diagonále je vidět čas lokálního přístupu do paměti. Na zbylé uzly přistupujeme vždy stejně kvalitní a dlouhou linkou QPI, takže časy jsou stejné.

Takto symetrický přístup k vzdáleným numa pamětem je méně častý. Většina velkých systému má topologii složitější, kde jsou propojovány uzly přes kontroléry a ty jsou pak přes další kontrolér propojené s dalšími uzlů. Dochází tak k přístupům k vzdálené paměti přes více numa uzlů a tím pádem dochází k pomalejšímu přístupu do vzdálené paměti. Teď se podívejme na obrázek

Jedná se o systém s 80 logickými jádry (40 fyzických) se zapnutou podporou hypertheading. Každý procesor této generace obsahuje 10 logických jader (5 fyzických). Každý procesor se svou lokální pamětí tvoří jeden numa



Obrázek 9: Systém se složitější numa topologii

uzel. CPU jsou po čtyřech spojené s jedním CPU kontrolérem. Ten je spojuje se zbylými 4 CPU přes další CPU kontrolér. Ze samotného obrázku je vidět, že nyní nebudou přístupy do pamětí dvojího druhu lokální a vzdálené, ale vzdálené se nám dále budou lišit podle toho přes kolik kontrolérů bude uzel přistupovat k paměti (1 nebo 2). Takže vidíme, že přístupy do pamětí budou dle Obrázku

Vypíšeme si nejdříve jak jsou jednotlivé jádra shlukovány do uzlů a poté se podíváme, jak se liší vzdálené přístupy do pamětí.

```
# numactl -H
available: 8 nodes (0-7)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9
node 0 size: 262133 MB
node 0 free: 250463 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19
node 1 size: 262144 MB
node 1 free: 256316 MB
node 2 cpus: 20 21 22 23 24 25 26 27 28 29
node 2 size: 262144 MB
node 2 free: 256439 MB
node 3 cpus: 30 31 32 33 34 35 36 37 38 39
node 3 size: 262144 MB
node 3 free: 255403 MB
node 4 cpus: 40 41 42 43 44 45 46 47 48 49
node 4 size: 262144 MB
node 4 free: 256546 MB
node 5 cpus: 50 51 52 53 54 55 56 57 58 59
node 5 size: 262144 MB
node 5 free: 256036 MB
node 6 cpus: 60 61 62 63 64 65 66 67 68 69
node 6 size: 262144 MB
node 6 free: 256468 MB
node 7 cpus: 70 71 72 73 74 75 76 77 78 79
node 7 size: 262144 MB
node 7 free: 255232 MB
```

A nyní se podíváme na přístupy do paměti mezi jednotlivými numa uzly.

```
node distances:
node 0 1 2 3 4 5 6 7
0: 10 12 17 17 19 19 19 19
1: 12 10 17 17 19 19 19 19
2: 17 17 10 12 19 19 19 19
3: 17 17 12 10 19 19 19 19
4: 19 19 19 19 10 12 17 17
5: 19 19 19 19 12 10 17 17
6: 19 19 19 19 17 17 10 12
7: 19 19 19 19 17 17 12 10
```

Nyní přístupy do paměti nabývají těchto hodnot:

1. Hodnoty 10 nejlepšího tedy nejnižšího času jsme dosáhli při přístupu do lokální paměti.

2. Hodnoty 12 druhého nejlepšího času přístupu jsme dosáhli při přístupu na vedlejší uzel, bez potřeby komunikovat přes numa kontrolér.

3. Třetí hodnoty jsme dosáhli při přístupu na vzdálený uzel, který komunikuje přes jeden kontrolér.

4. Nejhorší možný čas přístupu do paměti jsme dosáhli v případě, že komunikují dva numa uzly, které jsou propojeny pomocí dvou kontrolérů.

Při vyvažování u numa systému je nutné mít na paměti, že ne všechny procesory mají přístupy do všech segmentů pamětí stejně rychlé.
Numa vyvažování používá tři principy přesunu pro docílení vyváženosti:

1. Proces následuje svou paměť přesouvá se pouze proces – malé přesuny dat, obvykle levnější.

2. Paměť přesouváme za úlohou – kopírovaní obrovských bloků paměti může vést k časovým prodlevám.

3. Kombinace předchozích metod. Kód plánovače (v tomhle případě spíše mluvíme o funkcích na vyvažování jako o vyvažovači) zváží zdali je výhodnější přesouvat paměť nebo přesouvat proces.

Přístupy	Úloha A	Úloha B
Uzel 0	12	1022
Uzel 1	88	29
Uzel 2	994	14
Uzel 3	14	44

Tabulka 1: Přístupy na numa uzly

Plánovač si vede informace o přístupech úlohy do různých segmentů paměti (na různé numa uzly). Jsou registrovány přístupy do segmentů do kterých se přistupuje opakovaně, o implementaci sběru statistik přístupů si povíme později.

Úloha umístit úlohu na nejvhodnější pozici, nemusí být zcela tak jednoduchá. Vše komplikují sdílené data (třeba knihovna glibc), na které obvykle přistupuje více úloh.

Výběr kde umístit úlohu probíhá dle následujícího algoritmu. Pro každou úlohu A testuje numa uzel N

1. kontroluje zdali je uzel N lepší než uzel současně běžící úlohy (C). Úloha má větší množství svých dat na uzlu N než na současném uzlu C

2. jestliže platí bod 1. pak pro každé CPU na uzlu N zjišťujeme

   - zdali je CPU bez přidělené úlohy a můžeme tam umístit úlohu
   - když není CPU bez přidělené úlohy, zdali je lepší současnou úlohu odebrat z uzlu C
   - je přínos přesunu úlohy na uzel N větší než nevýhody přesunu.

4.3 Spuštění numa vyvažování

4.3.1 Periodické spouštění vyvažování při obsluze časovače přerušení

Tento způsob spuštění vyvažování úloh je implementován přesně tak, jak je popsáno v kapitole o vyvažování UMA SMP systému. Funkce `scheduler_tick` volá funkci `trigger_load_balance`. Ta vyvolá softwarové přerušení, které vyvolá obsluhu přerušení `run_rebalance_domains`. `run_rebalance_domains` volá funkci `rebalance_domains`, která prochází hierarchicky domény a vyvažuje skupiny v doméně pomocí volání funkce `load_balance`. Podrobněji se tomu věnujeme v kapitole 4.1.3 o aktivním vyvažování.

4.3.2 Vyvažování spouštěné jako ošetření page fault vyjímky

Aby mohl vyvažovací algoritmus rozhodnout, kde je vhodné umístit úlohu určenou k běhu, vede si statistiky přístupů na numa uzly každé úlohy. Sběr statistik je realizován tak, že jsou znepřístupněné určité stránky paměti a při přístupu úlohy do znepřístupněné paměti je vyvolána page fault vyjímka. Jako ošetření vyjímky je volána funkce `do_numa_page`.

Tato funkce volá funkci `task_numa_placement`, která má za úkol procházet statistiky přístupů do jednotlivých uzlů a vybere uzel do kterého měla úloha nejvíce přístupů. Ve funkci `task_numa_placement` je volána funkce `numa_migrate_preferred`, která uloží vybraný uzel do proměnné `numa_preferred_nid` (ze struktury `task_struct`). Na základě znalosti preferovaného numa uzlu je volána funkce `task_numa_find_cpu`, která vybere nejvhodnější CPU k běhu na vybraném uzlu (`numa_preferred_nid`). Funkce `numa_migrate_preferred` pak volá funkci `task_numa_migrate`, která zajistí přesun úlohy na vybrané CPU.

V této kapitole jsme si vysvětlili, že plánovač umisťuje úlohy a vyvažuje zatížení na numa systémy dle různých algoritmů (v prvním případě podle funkce `load_balance` ve druhém dle logiky v `task_numa_placement` a `task_numa_find_cpu`).

Jelikož jsou algoritmy pro výběr a zátěže úloh značně rozdílné je třeba testovat, zdali nenastane situace, že je dle algoritmu z `task_numa_find_cpu` a `task_numa_placement` úloha umístěna na nějaké CPU, která je následně přesunuta algoritmem `load_balance` při periodickém spuštění na jiné CPU. Přesuny úloh způsobují nedostupnost dat ve vyrovnávacích paměti, výměnu obsahu registrů, nebo dokonce kopírování pamětí mezi numa uzly. Pokud tato situace nastane dochází k značnému poklesu propustnosti daného systému.

4.4 Logika vyvažování na numa systémech v příkladech

Vyvažovací algoritmus na numa systémech je lepší si demonstrovat na příkladech. Plánovač sbírá statistiky přístupů jednotlivých úloh na numa uzly a ví také z front běžících úloh, kde je úloha umístěná. Na základě těchto informací dochází k rozmístění úloh na systému, tak jak je to demonstrováno v následujících příkladech.

Příklad 1 na umístění úlohy na CPU a numa uzlu na kterém neběží žádné jiné úlohy

Uzel	CPU číslo	Úloha
Uzel 0	0	A
Uzel 0	1	B
Uzel 1	2	žádná
Uzel 1	3	žádná

Tabulka 2: Tabulka lokací běžících úloh

Uzel	Úloha A	Úloha B
Uzel 0	30%	60%
Uzel 1	70%	40%

Tabulka 3: Statistiky přístupů do paměti

Z tabulky

Přesunem jedné úlohy z uzlu 0 na uzel 1 by byl vyřešen také problém s vyvážením. Je žádoucí, aby byly úlohy rozmístěné rovnoměrně na každé úrovni domén, v tomto příkladě to znamená, po jedné na úrovni numa uzlů. Tento příklad je zcela jednoduchým příkladem, kdy máme dvě nezávislé úlohy. Kdyby šlo o závislé úlohy, přístupy do segmentů pamětí by mohly mít stejné obě úlohy.

Příklad 2 na výměnu úloh na numa uzlech

Uzel	CPU	Úloha
0	0	A
0	1	žádná
1	2	B
1	3	žádná

Tabulka 4: Tabulka lokací běžících úloh

Uzel	Úloha A	Úloha B
Uzel 0	30%	40%
Uzel 1	70%	60%

Tabulka 5: Statistiky přístupu do paměti

Úlohy jsou na úrovni domén numa uzlů vyváženy. To znamená, že v tomto příkladě běží na každém numa uzlu právě jedna úloha. Dále plánovač zjišťuje, zdali jsou úlohy a data na stejném numa uzlu, čili prověřuje, zdali úloha nemusí zbytečně přistupovat do vzdálených segmentů pamětí. Tady vidíme, že ačkoli úloha A běží na uzlu 0 většinu přístupů (70%) dělá do vzdálené paměti (na uzel 1). Nyní se podíváme, kde běží a kde přistupuje úloha B. Ta běží na uzlu 1 a přistupuje nejvíce (60% přístupů) do uzlu 1. Nyní abychom zjistili zdali má smysl výměna úloh musíme zjistit co by nám přineslo přehození úloh A a B mezi sebou (jelikož víme, že úlohy jsou rovnoměrně rozmístěny na doméně numa uzlů).
Sumarizací lokálních přístupu zjistíme jak jsme na tom před přehozením úloh.
Úloha A = 30 Úloha B = 60
Celkově tedy do lokální paměti máme 90/200.
Nyní stejný případ s prohozením úloh

Uzel	Úloha A	Úloha B
Uzel 0	30%	40% (lokální)
Uzel 1	70% (lokální)	60%

Tabulka 6: Statistiky přístupu do paměti

Úloha A = 70 Úloha B = 40
Celkově tedy přístup do lokální paměti máme 110/200. Jelikož je po přehození úloh numa doména vyvážená a přístupy jsou většinou do lokální paměti může dojít k přehození úloh. Před přehozením úloh se spočítá zdali cena přesunu

je nižší než cena ponechání úlohy a přistupování úlohy do vzdálené paměti.

Kroky algoritmu při vyvažování na numa uzlech

1. Plánovač zjišťuje, zdali je systém v numa doméně vyvážený (tj. každá skupina v doméně má přibližně stejné množství běžících úloh).

2. Poté plánovač sleduje, jestli jsou přístupy úloh do paměti lokální. V případě, že úlohy přistupují většinou do lokální paměti a množství procesů na doméně vyvážené, algoritmus skončí.

3. Dále plánovač hledá takové kombinace rozmístění úloh, které jsou na doméně vyvážené a většinu přístupů do paměti je lokálních.

4. Pro nejlépe ohodnocené kombinace rozmístění úloh, zjišťuje, zdali budou ceny migrace nižší než cena ponechání úloh na současném místě a běhu úloh s vzdálenou pamětí.

5. Nakonec dochází k samotnému přesunu úloh nebo pamětí dle nejvýhodnějšího ohodnocení z předchozího bodu

U vyvažování na úrovni numa uzlů se dále řeší grupování za účelem plánování celé skupiny úloh, které například sdílí stejné knihovny a ty jsou v paměti na numa uzlu, kde běží tato skupina úloh.

5 Testování

5.1 Testovací úloha

Pro vytvoření úloh, kterými zatížíme testovaný systém, použijeme modifikovaný benchmark linpack. Do originální verze linpacku byly přidány semafory, aby bylo možno workload úloh spustit v jeden okamžik a funkce na sběr statistiky běhu benchmarku. Benchmark linpack dělá aritmetické operace (násobí matice) a výsledkem je množství floating point operací za vteřinu.

5.2 Úrovně testování plánovače

5.2.1 Testování plánování fronty úloh na jednom jádru

Testování na jednom jádru provádíme tak, že spustíme v jeden okamžik na jednom jádru řadu úloh, a na konci se podíváme, jak dlouho jednotlivé úlohy běžely a jakých výsledků jsme dosáhli. Pokud bude plánovač spravedlivě přidělovat čas úlohám budou běžet stejně dlouho, množství výměn obsahu bude přibližně stejné pro každou úlohu, dosažená hodnota operací za sekundu bude také přibližně stejná. Jelikož jsme úlohy spouštěli na jádru se specifikovanou CPU afinitou, prověříme také, zdali úlohy běžely jen na jádru námi specifikovaném.

Pro testování plánování fronty úloh na jednom CPU vytvoříme frontu úloh (například 20 úloh) a spustíme je se stejnou afinitou CPU (to znamená, že všechny úlohy poběží na námi specifikovaném jádru) pomocí systémové utility taskset.

Poté zkontrolujeme ve výstupu benchmarku následující:

- Zdali úlohy opravdu běžely na jádru námi specifikovaném při startu úloh (v našem příkladu jádro s číslem 38)
- Zdali všechny úlohy běžely přibližně stejně dlouho, zdali počet výměn obsahu na CPU je přibližně stejný a zdali jsme dosáhli přibližně stejné hodnoty operací na plovoucí řádce (kflops) pro každou instanci benchmarku.
- V mpstat výstupu zkontrolujeme zdali byl procesor s frontou úloh neustále plně vytížen (viz mpstat zkrácený sumarizační výstup)

Výpis běhu benchmarku linpack

```
   times are reported for matrices of order  1000
     dgefa      dgesl      total       kflops     unit      ratio
 times for array with leading dimension of 1001
      2.89       0.01       2.90     230500       0.01      51.80
      2.17       0.01       2.17     307678       0.01      38.81
      2.82       0.01       2.83     236481       0.01      50.49
      2.21       0.01       2.22     301454       0.01      39.61
 times for array with leading dimension of1000
      2.04       0.01       2.04     327506       0.01      36.46
      2.42       0.01       2.42     275854       0.01      43.29
      2.14       0.01       2.14     311905       0.01      38.28
      2.13       0.00       2.13     313553       0.01      38.08
Unrolled Double  Precision 301454 Kflops ; 10 Reps

Nodename:       intel-canoepass-02.lab.eng.rdu.redhat.com
Sysname:        Linux
Release:        2.6.32-431.el6.x86_64
Version:        #1 SMP Sun Nov 10 22:19:54 EST 2013
Machine:        x86_64
----------------------------------------------------------------
Started on CPU 38.
Waiting for semaphore_id 9338882 to reach 0.
Started at 05-27-2014  13:50:05.452249
Unrolled Double Precision Linpack

     norm. resid      resid           machep         x[0]-1
       9.5        4.22017976e-12  2.22044605e-16  1.09912079e-13
Finished at 05-27-2014  13:51:03.855331
Time elapsed:   58.403 seconds
User time:      58.255 seconds
System time:    0.013 seconds
----------------------------------------------------------------
Page reclaims:  544
Page faults:    0
Voluntary context switches:     0
Involuntary context switches:   5912
----------------------------------------------------------------
PUs, ABSOLUTE_COREs, RELATIVE_COREs (CORE number inside SOCKET), SOCKETs
and numa nodes on which Benchmark-utils/linpackd was running.
ABSOLUTE_COREs represent logical numbering of COREs. It's horizontal
index in the whole list of COREs.  -1 means that the object could not be identified.
----------------------------------------------------------------
PUs:            38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
ABSOLUTE_COREs: 14 14 14 14 14 14 14 14 14 14
RELATIVE_COREs: 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
SOCKETs:        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
numa nodes:     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

MPSTAT zkrácený sumarizovaný výpis

První číslice na každém řádku označuje číslo PU (logická jednotka) další čísla znamenají procentuální vytížení daného PU aktualizované po pěti vteřinách (celkem 12číslic to znamená, že vytížení sledujeme jednu minutu). Dvě hvězdičky znamenají 100% vytížení.

Obrázek 10: mpstat zkrácený sumarizovaný výpis

```
 0  0  0  0  0  0  0  0  0  0  0  0  0
 1  0  1  0  0  0  0  0  0  0  0  0  0

                .    .    .

36  0  0  0  0  0  0  0  0  0  0  0  0
37  1  1  0  1  0  0  0  0  0  0  0  0
38  99 ** ** ** ** ** ** 99 ** ** ** 78
39  0  0  0  0  0  0  0  0  0  0  0  0
40  0  0  0  0  0  0  0  0  0  0  0  0

                .    .    .

46  0  0  0  0  0  0  0  0  0  0  0  0
47  0  0  0  0  0  0  0  0  0  0  0  0
```

Výsledek testování plánování fronty úloh

- Úlohy běžely na PU číslo 38, které jsme specifikovali při startu úloh.
- Všechny úlohy běžely přibližně 58 sekund, u všech úloh proběhlo přibližně 5900 výměn obsahu na CPU, hodnota dosažená benchmarkem byla u všech úloh přibližně 300,000 Kflops.
- Z mpstat výstupu vidíme, že PU 38 bylo po celou dobu běhu úloh maximálně vytížené.

Jelikož byly splněny všechny naše předpoklady říkáme, že plánovač plánuje férově úlohy z fronty úloh na jednom jádru.

5.2.2 Testování plánování úloh na SMP UMA systémech

Na SMP systémech testovací zátěž spouštíme dvěma způsoby:

1. Spouštíme úlohy se specifikovanou CPU afinitou. Nejprve analýzou topologie zjistíme optimální rozložení zátěže pro jednotlivá CPU. Za pomocí systémové utility taskset pak specifikujeme každé jedné úloze CPU afinitu.

2. Spuštění a vyvažování úloh řízené plánovačem. V tomto případě nespecifikujeme afinitu úlohám vše necháme na plánovači úloh.

Následně porovnáváme jakého výsledku jsme dosáhli u každého typu běhů. Podíváme se zdali plánovač, zbytečně nepřehazoval úlohy mezi jádry, porovnáváme časy běhů úloh, porovnáváme jakou hodnotu Kflops jsme dosáhli..

5.2.3 Testování plánování úloh na numa systémech

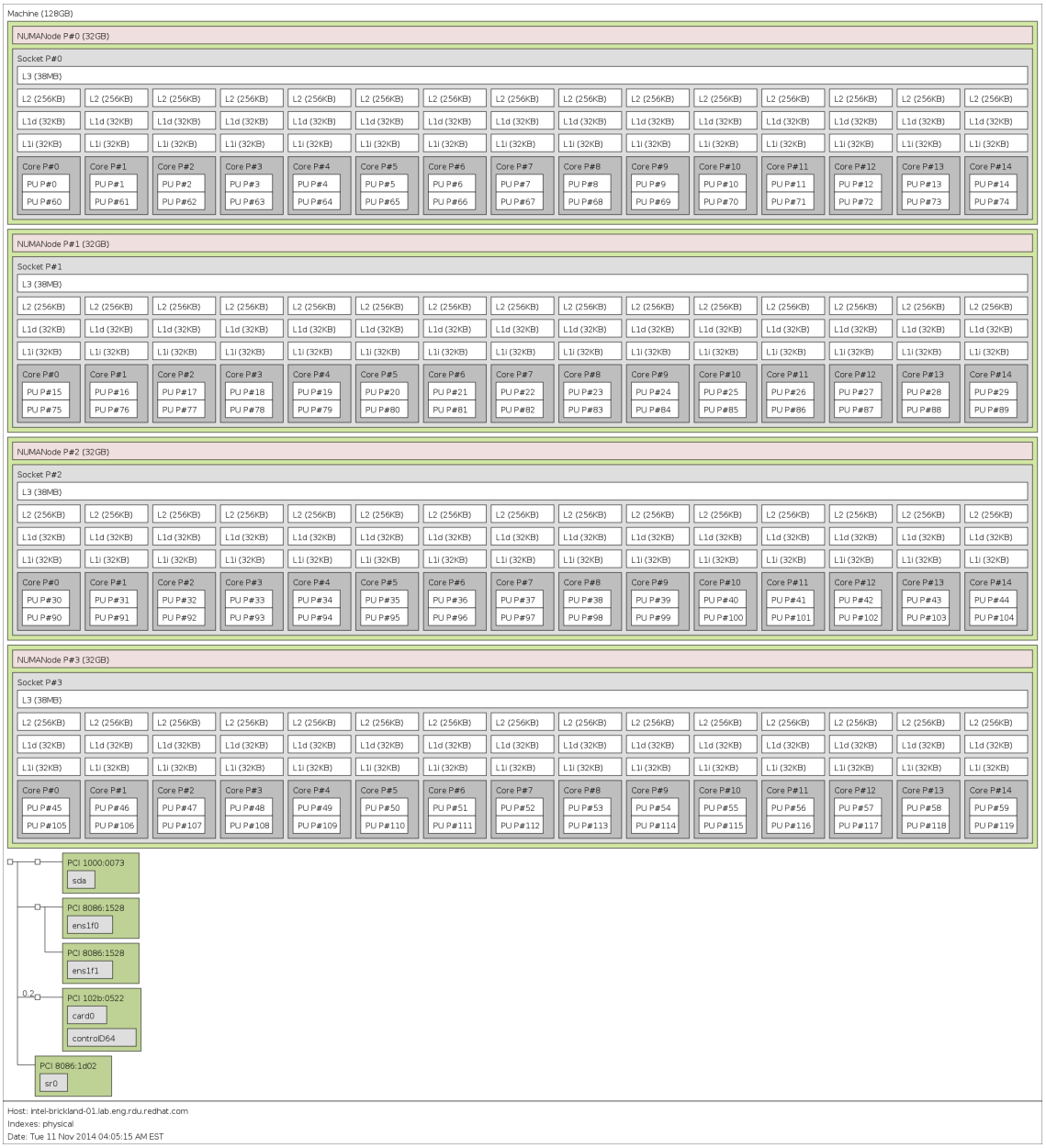
Jelikož jsou numa systémy variantou SMP systémů, bude se testování lišit jen tím, že úlohy budeme spouštět ještě třetím způsobem a to se specifikovanou numa afinitou. numa afinitu úloh specifikujeme za pomocí systémové utility numactl.

Pro naše testování vyvážení použijeme jednoduchý scénář. Budeme spouštět od jedné až po tolik na sobě nezávislých úloh kolik máme logických jader v systému (kvůli úspoře času při běhu testu zvolíme si vhodné krokování například po 4 úlohách). Jelikož je systém vybaven HT a každá dvě jádra mají jen jedno ALU, dá se předpokládat, že se výsledky budou zhoršovat přibližně od běhu, kdy budeme mít počet jader / 2 úloh spuštěných najednou. Jelikož v praxi běží ještě na procesoru některé systémové úlohy, je pravděpodobné, že uvidíme propad už dříve než u počet jader / 2 úloh spuštěných současně. Očekáváme, že plánovač rozmístí na každou logické jádro maximálně jednu úlohu a že všechny úlohy budou běžet paralelně. Pro prezentaci výsledků použijeme tentokráte grafy.

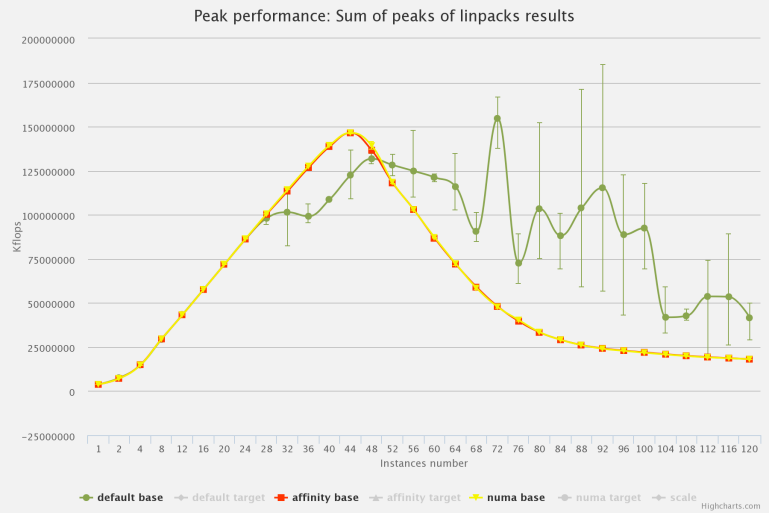
Během testování porovnáme sadu tří výsledků (plánovačem řízené, se specifikovanou CPU afinitou a se specifikovanou numa afinitou) s další sadou tří výsledků spuštěných na jiné verzi jádra. Záměrně volíme verzi jader, mezi kterými probíhal vývoj algoritmu pro numa vyvažování.

Nejdříve seběhneme linpack benchmark na starší verzi jádra operačního systému a to pro všechny 3 typy běhu (plánovačem řízené, s CPU afinitou, s numa afinitou). Pro každý typ běhu běžíme nejdřív 1 instanci linpacku, poté dvě zároveň, čtyři až nakonec 120 (což je celkový počet logických jader v systému viz. obrázek

Graf na obrázku



Obrázek 11: Testovaný systém se čtyřmi numa uzly a 120 HT jádry

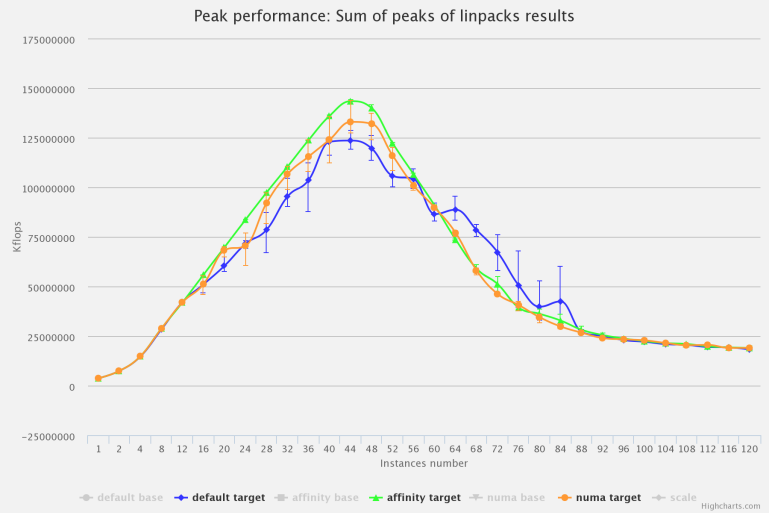


Obrázek 12: Výsledky linpacku na starším jádru

Z grafu na obrázku

Zjistíme, že běhy specifikované CPU afinitou se neliší vůbec, běhy specifikované numa afinitou se částečně liší a běhy řízené plánovačem se liší dost významně. Zjišťujeme, že kromě toho, že běhy na prvním jádru dosáhly vyšších hodnot kflops, tak mají také veliké rozptyly. Výsledek dosažený na prvním jádru plánovačem se může zdát velice dobrý, ale opak je pravdou. Jelikož jsme volili pro běhy se specifikovanou afinitou právě takové CPU a numa uzly, aby byl výsledek co nejlepší, je nemožné, aby plánovač rozmístil úlohy lépe a ty by dosáhly lepšího výsledku.

Nyní se podíváme na další grafy, které nám odhalí co se při testování stalo. Další graf (z obrázku

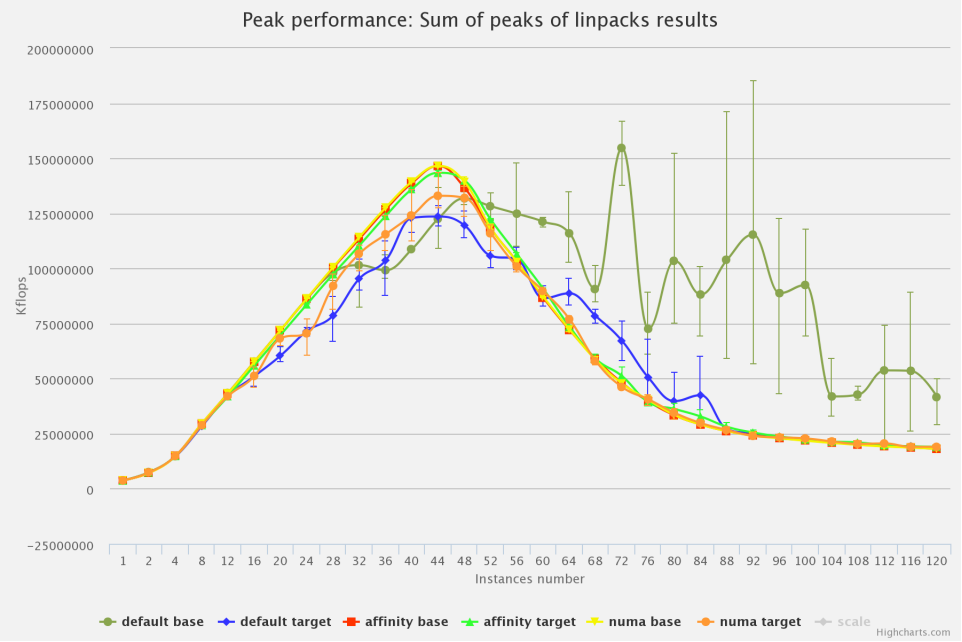


Obrázek 13: Výsledky linpacku na novějším jádru

Další graf (obrázek

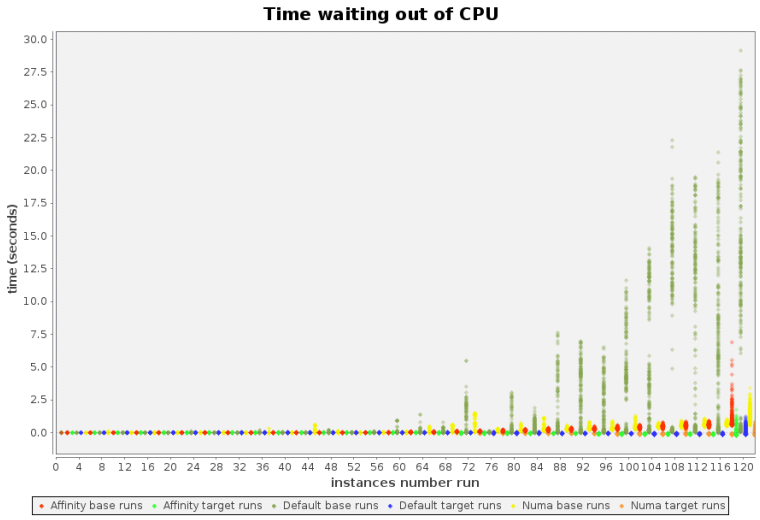
Další zajímavou informací může být zatížení jednotlivých CPU zjišťovaných utilitou mpstat.

Vybereme si plánovačem řízený běh, výsledky prvního testovaného jádra a počtu instancí 72. Zde totiž vidíme značné rozdíly oproti výsledkům dosaženým u druhého testovaného jádra. Výstup je zkrácený ukazuje jen prvních 20 PU. Je zde vidět, že dochází k přesunům úloh na jiná PU, některé úlohy jsou spuštěné později, nebo dokonce je jedna úloha spuštěna, až je druhá dokončena. Toto samozřejmě úloze nechá celou vyrovnávací paměť a úloha pak seběhne rychleji. Toto řešení nemá nicméně nic společného s férovostí (jedna úloha běží další čeká).

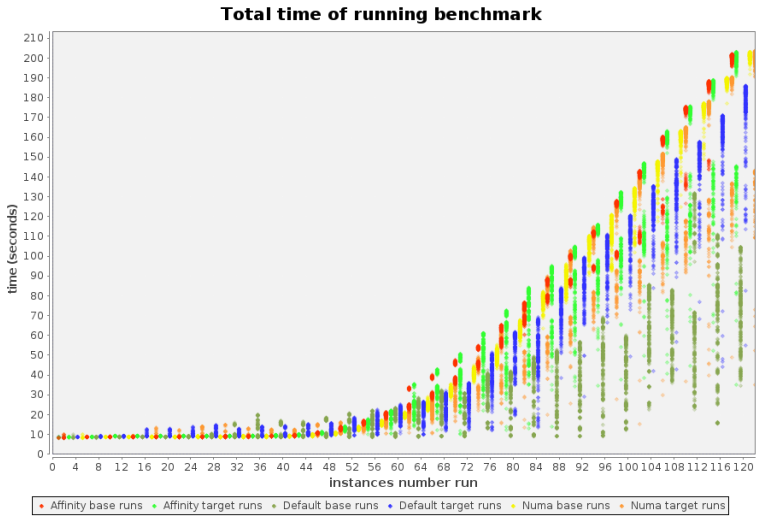


Obrázek 14: Výsledky ze dvou kernelu v jednom grafu

Dále nás zajímá kde plánovač umisťuje paměť procesu a kde samotný proces. K tomu účelu použijeme perf bench benchmark. Tento příklad ukazuje nevhodné kopírování paměti mezi numa uzly. Napravo na obrázku 15 je paměť úloh, nalevo je zobrazeny rozmístění úloh na numa uzlech. V tomto příkladu jsou spuštěny 2 vlákna celkem mají 8GB paměti a běží 20 min. Až po 6ti minutách dochází k rozdělení dat a procesů na 2 numa uzly.



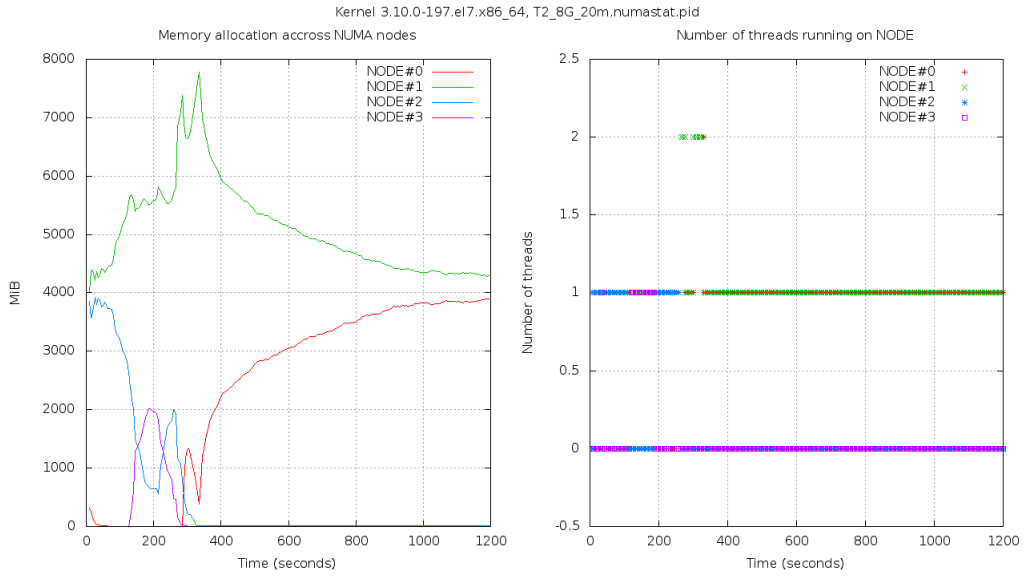
Obrázek 15: Čas strávený úlohami mimo CPU



Obrázek 16: Celkový čas běhu úloh

Obrázek 17: mpstat zkrácený sumarizovaný výpis

```
 0  1  0  1  0  0  0 49 ** 85  0 19 ** ** ** 85
 1  1  0  0  0  0  0 47 ** ** ** 90 ** ** ** 56
 2  0  0  0  0  0  0 37 ** ** ** 79 ** ** ** 79
 3  0  0  0  0  0  0 50 ** ** ** 86 76 ** ** **
 4  0  0  0  0  0  0 30 ** ** ** 92  0  0  0  0
 5  0  0  0  1  0  0  4 ** ** ** ** 16  0  0  0
 6  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 7  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 8  1 ** ** ** ** ** 52  0  0  0  0  0  0  0  0
 9  1 ** ** ** ** ** 54  0  0  0  0  0  0  0  0
10  0 ** ** ** ** ** 48  0  0  0  0  0  0  0  0
11  0 ** ** ** ** ** 35  0  0  0  0  0  0  0  0
12  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
13  0 ** ** ** ** ** 46  0  0  0  0  0  0  0  0
14  0 ** ** ** ** ** 27  0  0  0  0  0  0  0  0
15  1  1  1  0  0  1  0  0  3 ** ** ** ** 89 **
16  1 ** ** ** ** ** ** ** 97 ** ** ** 95 78 **
17  1  1  0  0  0  0  1  0  0 96 ** ** ** 81 **
18  1 ** ** ** ** ** ** ** 99 82 ** ** ** 18  0
19  0 ** ** ** ** ** ** ** 86 80 ** ** ** 23  0
20  1 ** ** ** ** ** ** ** 82  0  0  0  0  0  0
```



Obrázek 18: Graf znázorňující současný běh dvou úloh

6 Závěr

Po bližším obeznámení se s plánovačem úloh jsem došel k přesvědčení, že návrh plánovače byl precizně navrhnut a implementován po numa vyvažování. Připadá mi, že numa vyvažování nebylo přidáno do kódu v duchu návrhu celého plánovače, hlavně tedy třídy pro plánování obyčejných úloh. Kód numa vyvažování je dlouhý, postrádám v něm jakýsi koncept, vadí mi heuristika. Překvapilo mě, že se při numa vyvažování kalkuluje s pomocí konstant jako například výpočetní síla skupiny procesorů, nebo jak získává historii přístupu na stránky. Tu určí tak, že dosavadní přístupy vydělí dvěma. Jsem přesvědčen, že tvůrci numa vyvažování udělali chybu v tom, že se numa vyvažování realizuje dvěmi různými funkcemi, které si nejsou vůbec podobné a spouštějí při dvou událostech (jedna periodicky, druhá při přístupu na uzamčené stránky paměti). To očividně vede k neustálému migrování úloh a jejich dat z jednoho numa uzlu na druhý. Od vývojáře numa plánovače jsem byl informován, že toto je slabina současného plánovače a že se nyní vymýšlí, jak tuto slabinu odstranit.

Reference

[1] *Volker Seeker:* Process scheduling in Linux
University of Edinburgh, Critical blue 2013
Ω_namelabel_name http://criticalblue.com/news/wp-content/uploads/2013/12/linux_scheduler_notes_final.pdf

[2] *Dokumentace na www.kernel.org k linuxovému plánovači*
`https://www.kernel.org/doc/Documentation/scheduler/`

[3] *IBM developerworks článek*
`http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/`

[4] *Článek z linuxjournal*
`http://www.linuxjournal.com/magazine/completely-fair-scheduler`

[5] *Emaily věnované plánování z lwn.net a lkml.org*
`https://lkml.org`
`https://lwn.net`

[6] *Prezentace od Rika van Riela*
Interní RedHat prezentace od jednoho z vývojářů vyvažovacího algoritmu v CFS plánovači