Výpočetní cluster na platformě Linux

Computing cluster on Linux platform

Bc. Tomáš Bezděk

Diplomová práce
2009

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Tomáš BEZDĚK**
Studijní program: **N 3902 Inženýrská informatika**
Studijní obor: **Počítačové a komunikační systémy**

Téma práce: **Výpočetní cluster na platformě Linux**

Zásady pro vypracování:

1. **Zpracujte rešerši na zadané téma.**
2. **Otestujte praktické nasazení a nárust výkonu.**
3. **Připravte cluster pro nasazení v podmínkách UTB.**
4. **Definujte požadavky HW a infrastruktury na provoz clusteru.**

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **The Linux Clustering Information center [online]. [2004] [cit. 2009-01-26]. Dostupný z WWW: [http://lcic.org/].**
2. **Kerrighed [online]. 2008 [cit. 2009-01-26]. Dostupný z WWW: [http://www.kerrighed.org/wiki/index.php/Main_Page].**
3. **LOTTIAUX, Renaud. OpenMosix, OpenSSI and Kerrighed [online]. 2006 [cit. 2009-01-26]. Dostupný z WWW: [http://hal.inria.fr/inria-00070604/en/].**
4. **SPECTOR, David. Building Linux Clusters. [s.l.] : OReilly, 2000. 332 s. ISBN 1565926250.**

Vedoucí diplomové práce: **doc. Ing. Martin Sysel, Ph.D.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2009**

Termín odevzdání diplomové práce: **27. května 2009**

Ve Zlíně dne 13. února 2009

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Diplomová práce se zabývá problematikou výstavby clusteru na platformě Linux. Čtenář získá informace z oblasti výstavby clusterů a jejich provozu. Součástí teoretické části je rovněž přehled možných použitelných řešení.

V praktické části práce je popsán výběr vhodných programových komponent pro realizaci a následně je popsána praktická realizace, která je zasazena do podmínek pro reálný provoz. Práce se věnuje i problematice programování paralelních aplikací a jejich provozování na OS Linux. Součástí práce je i návod na vytvoření clusteru.

Klíčová slova: Linux, cluster, Kerrighed

ABSTRACT

Graduation thesis deals with the cluster construction on Linux platform. Reader acquires information from the clusters construction and their operation. A part of the theoretical part is also overview of applicable potential solutions.

In the practical part of work is described the appropriate software components selection for the implementation and then the practical implementation is described that is set in real operation conditions. This work addresses the issue of programming parallel applications and their operation on the Linux OS. A part of the work is to create cluster instructions.

Keywords: Linux, cluster, Kerrighed

Tímto chci poděkovat vedoucímu diplomové práce panu doc. Ing. Martinu Syslovi, Ph.D. za odborné vedení, cenné rady a připomínky, které mi poskytoval při řešení této práce.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně ……………………
Podpis diplomanta

OBSAH

ÚVOD 9
I TEORETICKÁ ČÁST 10
1 PARALELNÍ ÚLOHA 11
1.1 ARCHITEKTURY SMP A NUMA 12
1.2 MPP (MASIVE PARALLEL PROCESSING) 13
1.3 GPGPU 13
1.4 VOLBA ARCHITEKTURY 14
2 TYPY CLUSTERŮ 15
2.1 HA (HIGH AVAILABILITY) CLUSTER 15
2.2 STORAGE CLUSTER 15
2.3 HPC (HIGH PERFORMANCE COMPUTING) CLUSTER 15
2.3.1 SSI (Single System Image) cluster 15
2.3.2 Beowulf 16
3 LINUX 17
3.1 DESIGN 17
3.2 SPRÁVA PROCESŮ A VLÁKEN V OS LINUX 18
3.3 ŽIVOTNÍ CYKLUS PROCESU 20
4 PROJEKTY PRO OS LINUX UMOŽŇUJÍCÍ VYTVOŘIT CLUSTER 21
4.1 BEOWULF 21
4.2 OPENMOSIX 21
4.3 OPENSSI 22
4.4 KERRIGHED 22
5 ARCHITEKTURA KERRIGHED 24
II PRAKTICKÁ ČÁST 27
6 NASAZENÍ CLUSTERU 28
6.1 VOLBA CLUSTER PROJEKTU A DISTRIBUCE 28
6.2 TECHNICKÉ ŘEŠENÍ 29
6.3 POUŽITÝ HARDWARE 29
6.4 POUŽITÍ 30
7 PROGRAMOVÁNÍ PARALELNÍCH APLIKACÍ 35
7.1 VÍCEPROCESOVÁ APLIKACE 35
7.2 VÍCEVLÁKNOVÁ APLIKACE 39
7.3 SYNCHRONIZAČNÍ PROSTŘEDKY 42
7.3.1 Semafor 42

7.3.2 Mutex 43
7.4 PROGRAMOVÁNÍ APLIKACE PRO KERRIGHED 43
8 TEST 45
8.1 VLASTNÍ TESTOVACÍ APLIKACE 45
8.2 PŘEKLAD PROGRAMŮ 48
9 ZÁVĚREČNÁ DOPORUČENÍ 50
9.1 DOPORUČENÍ PRO VÝBĚR HARDWARE 50
9.2 DOPORUČENÍ PRO VÝSTAVBU SÍTĚ 50
ZÁVĚR 51
ZÁVĚR V ANGLIČTINĚ 52
SEZNAM POUŽITÉ LITERATURY 53
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 55
SEZNAM OBRÁZKŮ 57
SEZNAM PŘÍLOH 58

ÚVOD

Počítačový cluster je skupina vzájemně propojených počítačů za účelem spolupráce na stejné úloze. Výsledkem spolupráce je, že tato skupina navenek vystupuje jako jeden výkonný počítač. Počítačové clustery jsou budovány z několika různých důvodů, motivací může být: zvýšení výpočetního výkonu, propustnosti nebo zvýšení dostupnosti systému v čase, přičemž jejich cena je nižší, než jediný počítač srovnatelných parametrů.

Pro výstavbu počítačových clusterů je s úspěchem využíváno převážně běžných počítačů třídy PC nebo víceprocesorových systémů propojených počítačovou sítí Ethernet. Ve zvláštních případech bývají využity k propojení technologie Infiniband nebo Myrinet propojující speciálně modifikované servery.

V poslední době začínají být počítačové clustery doplňovány GPGPU (General-Purpose computing on Graphics Processing Units) založenými na jádrech grafických karet v podobě nVvidia CUDA (Compute Unified Device Architecture) nebo AMD FireStream.

Tyto systémy nacházejí uplatnění pro provádění vědeckých výpočtů nebo simulací, například nárazových zkoušek automobilů nebo vývoje počasí a rovněž při zpracování velkých objemů dat. Další oblast pro aplikaci existuje například u telekomunikačních operátorů, nebo poskytovatelů služeb při zvyšování odolnosti proti výpadku.

Přestože problematika clusterů se zdá být složitá a málo využívaná tak i běžný uživatel denně využívá jejich služeb, ať už jde o vyhledávače jako například Google , mobilní a počítačové sítě nebo projekty jako SETI@home nebo BOINC, které umožňují jakémukoliv uživateli podílet se na výpočtu úkolu. [1]

I. TEORETICKÁ ČÁST

1 PARALELNÍ ÚLOHA

Základním cílem pro který jsou výpočetní clustery stavěny, je zkrácení doby provádění výpočtu. Toho je dosaženo pomocí rozdělení úlohy na několik nezávislých celků, které jsou zpracovány jednotlivými výpočetními jednotkami. Jednotlivé části jsou pak na výpočetních jednotkách vykonávány sekvenčně. Z toho vyplývá, že větší množství výpočetních jednotek než paralelních vláken v programu nepřináší žádný užitek. Doba běhu úlohy je pak teoreticky nepřímo úměrná počtu uzlů. V praxi je zvyšování výkonu omezeno prodlevami při přenosu dat a zpráv mezi jednotlivými částmi clusteru a tím, jak lze úlohu rozdělit.

Pro zvýšení výpočetního výkonu počítače bývá přistoupeno k možnosti instalovat do počítače více výpočetních jednotek (obvykle mikroprocesorů). Možnost osazení více mikroprocesorů je doménou zejména pro výkonné pracovní stanice a servery. Pro použití v osobních počítačích a přenosných počítačích byly vyvinuty technologie integrování více mikroprocesorů do jednoho pouzdra, nebo rovnou na jeden kus křemíkového substrátu. Jde o takzvané vícejádrové mikroprocesory. Obě výpočetní jádra takového procesoru komunikují s periferiemi za pomoci společné sběrnice. Tento přístup umožnil ve srovnání s víceprocesorovými systémy poskytnout stejný výpočetní výkon při mnohem nižších nákladech. Snížení nákladů bylo dosaženo tím, že složitost základní desky pro vícejádrový procesor je prakticky stejná jako pro jednojádrový procesor.

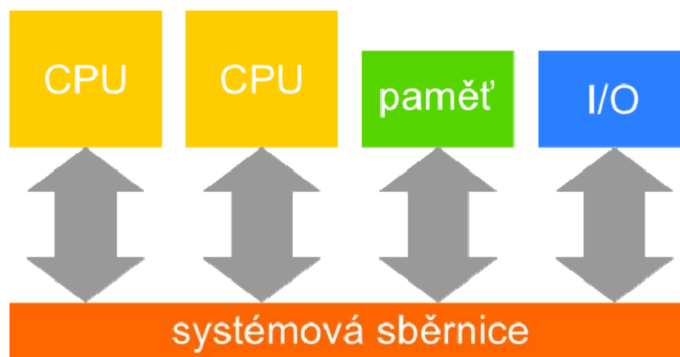
Pro výstavbu víceprocesorových systémů jsou obvykle dostupné základní desky pro 2 procesory, výjimečně pro 4 nebo až 8 procesorů. Při výstavbě systémů s vyšším množstvím procesorů jsou využity specializované systémy, jejichž cena je velmi vysoká, a velmi rychle zastarávají. Levnější alternativou je zvyšovat výkon pomocí spojování běžných osobních počítačů a serverů do clusteru.

Aby bylo takového systému možné efektivně využít, je nutná jeho podpora v operačním systému a spouštěné aplikaci.

U víceprocesorových počítačů existuje několik architektur, které definují způsob spolupráce výpočetních jednotek. Mezi architektury využívající sdílenou paměť patří SMP (Symmetric MultiProcessing) a NUMA (Non-Uniform Memory Access), naproti tomu MPP (Massive Parallel Processing) je architektura využívající distribuovanou paměť. [1],[2]

1.1 Architektury SMP a NUMA

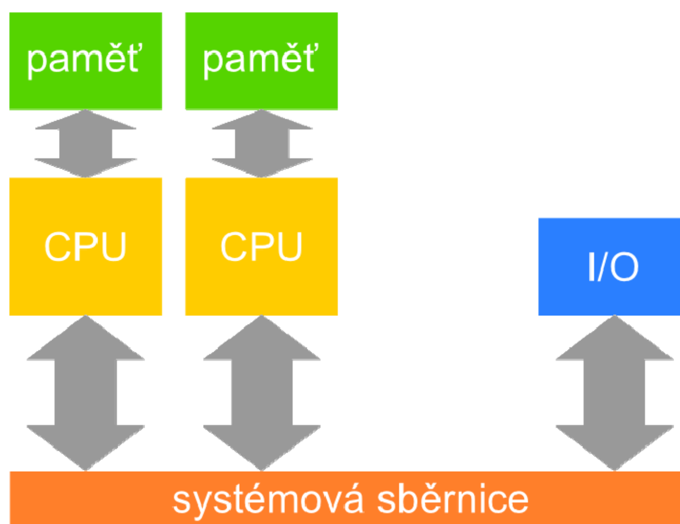
SMP, jinak také symetrický multiprocesing, je architektura u které je dva a více mikroprocesorů připojena ke společné sběrnici, pomocí níž komunikují s pamětí a periferiemi, které spolu navzájem sdílí jak je naznačeno na obrázku (*Obr. 1*). Jedná se o architekturu, která byla vyvinuta zejména pro menší počítače, které jsou osazeny obvykle ne více než 8 procesory.



Obr. 1: Blokové schéma typického SMP systému

Vzhledem k tomu, že všechny procesory paměť sdílí, je nutné zpracovávat požadavky o přístup. Požadavky o přístup do paměti a k periferiím jsou řazeny do FIFO (First In First Out) fronty a u SMP dochází ke zpoždění přímo úměrnému počtu použitých procesorů. SMP je tedy využíváno pro maximálně 8 procesorů.

Modifikováním architektury SMP vzniká architektura NUMA. Hlavní úprava spočívá v připojení paměti přímo k procesoru jak je naznačeno na obrázku (*Obr. 2*).



Obr. 2: Blokové schéma systému NUMA

Tato modifikace poskytuje procesoru mnohem vyšší rychlost přístupu do paměti, která je k němu připojena přímo. Samozřejmostí je, že může přistoupit i do paměti, která je k systému připojena prostřednictvím jiného procesoru, ale v takovém případě je přenos dat a odezva výrazně pomalejší ve srovnání s přístupem paměti připojené přímo.

SMP nebo NUMA využívají počítače nebo servery osazené více procesory (popřípadě vícejádrovými procesory) nebo některé clustery. Fungují tak, že operační systém se stará o migraci procesů, nebo jejich vláken mezi procesory a také sdílení paměti (nebo jejích stránek). Spuštěné programy pak jsou přesouvány buď mezi procesory v rámci jednoho počítače, nebo rovnou mezi jednotlivými počítači v případě clusteru. Výhodou je, že spouštěné programy mají k dispozici veškerou dostupnou paměť všech počítačů clusteru. [2],[3]

1.2 MPP (Masive Parallel Processing)

Systémy, které využívají MPP přístup nesdílejí žádné prostředky. Jedná se o autonomní jednotky, které se samostatně podílejí na své části úkolu. Výhodou je, že systém může být silně heterogenní z pohledu použitých architektur jednotlivých uzlů i z pohledu použitých operačních systémů. Projekty využívající tento přístup umožňují programátorům nahlížet na heterogenní prostředí jako na jeden paralelní počítač s množstvím výpočetních jednotek specializovaných pro různé typy úloh. Tyto nástroje jsou využívány již ve fázi programové realizace úlohy a řeší zejména předávání zpráv a dat mezi jednotlivými částmi úlohy.

Tato architektura je vhodná například pro simulaci vývoje počasí, nárazových testů vozidel nebo biochemických procesů a bývá realizována pomocí clusterů. [4]

1.3 GPGPU

General Purporse computing on Graphic Processing Unit je zvláštním případem použitelným pro náročné výpočetní úlohy. Tato architektura je velmi blízká MPP a ke svému chodu využívá grafického procesoru. Současné grafické karty obsahují na rozdíl od předchozích generací unifikované jednotky (tzv. shadery) namísto několika specializovaných skupin. Tyto jednotky jsou specializovány na provádění úzké skupiny výpočtů s plovoucí čárkou (floating point operation) zaměřených na grafické operace. Velkou výhodou je, že úzká skupina výpočtů může být prováděna mnohem rychleji než na CPU (Central Processing Unit), který je vhodný spíše na provádění obecných

výpočtů. Implementace této technologie probíhá pomocí knihovny například pro jazyky C, C++. V současné době je tato technologie dostupná na grafických kartách výrobců AMD a nVidia. [5]

1.4 Volba architektury

Na rozhodnutí, jaký typ přístupu bude zvolen má vliv typ aplikace, která bude na clusteru provozována. V případě, že aplikace je vyvíjena provozovatelem clusteru, nebo provozovatel má přístup ke zdrojovým kódům, je možné přistoupit k řešení MPP. Jedná se o sadu nástrojů, která pomůže s rozdělením aplikace na dílčí celky. Pokud nastane situace, kdy se jedná obecně o aplikace do nichž není možné provádět zásahy, nebo o spoustu různých menších aplikací, je vhodnější využít SMP případně NUMA, protože tady řeší rozložení zátěže právě operační systém.

Zjednodušeně řečeno technika MPP je integrována do provozované aplikace a technika SMP a NUMA vyžaduje podporu operačního systému.

2 TYPY CLUSTERŮ

Podle účelu, k němuž má být cluster určen bylo vyvinuto několik technik, které jsou využívány pro správu výpočetního výkonu. Existuje několik kritérií, podle nichž je možné clustery roztřídit. Následující rozdělení je zejména z hlediska primárního účelu a má vliv na technickou realizaci hardwarového a softwarového vybavení. Některá softwarová řešení jsou natolik komplexní, že umožňují realizovat více z následujících typů clusterů. [6],[7]

2.1 HA (High Availability) cluster

HA cluster se skládá z několika počítačů s identickým obsahem, kdy při výpadku počítače dojde k jeho zastoupení ostatními. Tento cluster bývá buď typu failover, kdy služba je spuštěna na jednom počítači a v případě výpadku je služba spuštěna na náhradním stroji a funkci převezme, nebo scalable, kdy služba běží na více počítačích a obsluha jednotlivých klientů je rovnoměrně rozdělována mezi všechny funkční počítače,díky využití techniky vyrovnávání zátěže tzv. load balancing.

2.2 Storage cluster

Tento typ clusteru je využíván k distribuovanému ukládání dat mezi jednotlivé stanice clusteru pro zajištění vysokého výkonu a spolehlivosti. Cíle je dosaženo využitím speciálních systémů souborů, které mají za úkol zajistit redundanci dat, rozložení zátěže a zamykání souborů. Princip funkce takového clusteru je velmi blízký funkci RAID (Redundant Array of Inexpensive Disks) úložiště, avšak nepracuje na úrovni blokového zařízení, ale na úrovni jednotlivých souborů.

2.3 HPC (High Performance Computing) cluster

2.3.1 SSI (Single System Image) cluster

Slouží pro sestavení velmi výkonného víceprocesorového stroje pomocí seskupení více počítačů, pro provozování běžných aplikací. Sestavený víceprocesorový stroj je pouze virtuální a jeho skutečná funkce je zajištěna úpravou jádra použitého operačního systému, které se stará o přesouvání spuštěných programů mezi jednotlivými fyzickými počítači. V tomto případě tedy nemusí být nutně provozována pouze jedna úloha, ale může být

provozováno velké množství menších aplikací. Použité počítače bývají obvykle střední cenové hladiny. Práce dále věnuje pozornost pouze tomuto typu.

2.3.2 Beowulf

Tato skupina clusterů bývá nazývána podle nejpopulárnějšího projektu v této kategorii. Jedná se o model určený pro paralelní výpočty, který staví na technologiích MPI (Message Passing Interface) a PVM (Parallel Virtual Machine), které mohou být provozovány i v silně heterogenním prostředí. Tento projekt se nezaměřuje na modifikaci operačního systému pro provozování paralelních výpočtů, ale jde o sadu nástrojů, které jsou využity při programové realizaci aplikace. To se sebou nese nemožnost použití v aplikacích, ke kterým není možné získat zdrojový kód. [8]

3 LINUX

Linux je jádro počítačových operačních systémů vyvíjené pod licencí GNU/GPL. Jeho vývoj začal v roce 1991 jako experiment finského studenta Linuse Torvaldse, později se do vývoje zapojilo větší množství nadšenců. V současné době vývojem do jádra přispívají velké společnosti jako RedHat Inc. Systém rovněž získává podporu na serverech od velkých společností jako IBM, Hewlett-Packard a Novell.

Původní vývoj byl zaměřen na 32 bitové procesory 386 a kompatibilní. V průběhu času byla zařazena podpora většiny 32 a 64 bitových architektur. Je tedy možné jeho nasazení v široké škále zařízení od mobilních telefonů až po superpočítače.

Protože Linux je pouze jádro operačního systému, je pro jeho použití jádro sdružit do distribuce s dalšími nástroji, knihovnami a programy. Většina použitelných nástrojů (bash, emacs, gcc, glib) byla vytvořena v rámci projektu GNU krátce před vytvořením jádra Linux. V distribucích bývá mimo aplikace uveřejněné pod licencí GNU/GPL jádro Linux
a nástroje z projektu GNU, proto distribuce bývají obvykle označovány jako GNU/Linux například: Debian GNU/Linux.

Pro výstavbu clusteru byl vybrán Linux z důvodu nízké ceny a licence připouštějící provádět v systému změny. Dalším důvodem je dostupnost kvalitních projektů umožňujících výstavbu clusteru a aplikací, které by bylo možné provozovat. [10]

3.1 Design

Operační systémy GNU/Linux jsou víceúlohové (multitaskové) systémy unixového typu s monolitickým jádrem. Monolitické jádro na rozdíl od mikrojádra se stará o správu procesů a komunikaci mezi nimi, správu paměti, síťovou vrstvu, periferie a ovladače dalších zařízení. Jeho základní principy vychází z OS Unix z 80. let 20. století. Na rozdíl od ostatních podobných operačních systémů (Solaris, QNX, Irix, Mac OS X) je Linux do jisté míry univerzální. To je dáno tím, že ostatní unixové systémy vznikaly přímo pro konkrétní účel, pro konkrétní málo rozšířenou platformu. Domovskou platformou Linuxu je x86 nicméně je portován na většinu 32 a 64 bitových platforem.

Možnou výhodou je, že Linux až na nepatrné odchylky splňuje standard POSIX (Portable Operating System Interface). Tato vlastnost velmi zjednodušuje vytváření nových aplikací

a stejně tak i jejich portování na jiné unixové operační systémy. Pochopitelně, že vývojář aplikace pro Linux není omezen pouze na rozhraní POSIX a může využít rozhraní specifického pro Linux, bohužel se tím komplikuje možnost portování na jiný operační systém.

První rané verze jádra Linux spoléhaly na kooperativní multitasking, který byl v pozdějších verzích nahrazen multitaskingem preemptivním. Zjednodušeně lze říct, že kooperativní multitasking spoléhá na spuštěné programy, které si mezi sebou navzájem předávají procesorový čas, naproti tomu u preemptivního multitaskingu jsou kvanta procesorového času přidělována programům pomocí tzv. plánovače (scheduleru), který je součástí jádra operačního systému.

Od Linux verze 2.0.0 (vydána 9.6.1996) byla začleněna podpora pro SMP a bylo tedy možné tento systém provozovat na víceprocesorových systémech. Mezi další průlomové verze patří i verze 2.2.0 a 2.4.0 (vydány 25.1.1999 a 4.1.2001) kdy bylo masivně rozšířeno množství podporovaného hardware a technologií. Z pohledu provozování paralelních aplikací přibyla velká inovace ve verzi 2.6.0 (vydána 17.12.2003) kdy byla přidána podpora vláken. Na jádrech verze nižší než 2.6.0 sice bylo možné provozovat vícevláknové aplikace, ale bylo nutné aplikaci vytvořit s podporou knihovny LinuxThreads. Tato implementace měla bohužel své nevýhody a nebyla slučitelná se specifikací POSIX Threads. Od verze 2.6.0 je dostupná implementace prostřednictvím knihovny NTPL (Native Posix Thread Library), která je známá pod názvem libpthread. [11]

3.2 Správa procesů a vláken v OS Linux

Mezi základní úkoly jádra systému patří správa běžících procesů a vláken. Do toho spadá jejich vytváření, plánování, nastavování, ukončování atd.

Zjednodušeně lze říct, že proces je běžící program. U OS Linux je proces objekt, který pracuje podle kódu programu, využívá svůj adresní prostor, využívá služby jádra a komunikuje s ostatními procesy. Vlákno je objekt pracující podle kódu programu, lišící se od procesu tím, že je jeho součástí a sdílí prostředky (adresní prostor paměti a prostředky jádra) s ostatními vlákny v procesu. Z pohledu jádra bývá využíván termín úloha (task). Jde o objekt, jehož kód je vykonáván sekvenčně. Každému vláknu v procesu odpovídá jedna úloha. V určitý časový okamžik běží na jednom procesoru nejvýše jedna úloha.

Úlohy běží na jednotlivých procesorech v rámci přidělených časových kvant. Úloha dostane přiděleno časové kvantum, a dokud jej nevyčerpá, není uspána, nebo nedojde k jeho odebrání, vykonává své programové instrukce.

Důležité je, že každé vlákno procesu je z hlediska plánování chápáno jako samostatná úloha, je tedy plánováno nezávisle na jiných vláknech téhož procesu.

Na plánování má zásadní vliv použitý plánovač. V jádru OS Linux jsou v současné době k dispozici tyto čtyři plánovače.

- **normální plánovač** (`SCHED_OTHER`) – pro většinu použití (běžné úlohy)
- **dávkový plánovač** (`SCHED_BATCH`) – stejně jako normální plánovač, ale s úpravou pro neinteraktivní úlohy
- **plánovač FIFO** (`SCHED_FIFO`) – pro úlohy pracující v reálném čase
- **plánovač round-robin** (`SCHED_RR`) – podobný jako FIFO

Vytvoření nového procesu je relativně jednoduchá operace. Již běžící proces zavolá funkci `fork()`, která zajistí, že proces je zkopírován a nadále běží ve dvou nezávislých exemplářích. Z pohledu jádra je situace složitější o identifikátor procesu, tzv. PID (Process IDentifier). Tento identifikátor je číslo z intervalu 1 až 32767, je automaticky inkrementováno, a identifikátor je přidělen vždy jen jednomu procesu. V případě, že je dosaženo maxima jsou volné identifikátory znovu použity pro nový proces. Přidělený PID je možné zjistit zavoláním funkce `getpid()`. Každý proces vyjma procesu **init** má svého rodiče (proces, kterým byl vytvořen).

Při vytváření nového procesu se alokované stránky paměti z pohledu procesu zkopírují. Ve skutečnosti ale nejsou stránky zkopírovány v okamžiku vzniku nového procesu. Dokud probíhá pouze čtení, pak zůstávají stránky společné pro starý i nový proces. Ke zkopírování dojde až v okamžiku, kdy se do dané stránky pokusí některý z procesů zapsat, nebo si zkopírování explicitně vynutí (například zamčením stránky). Tato metoda bývá nazývána copy-on-write.

Po skončení běhu procesu je nastavena návratová hodnota, kterou může kontrolovat rodičovský proces.

Práce s vlákny je velmi podobná práci s procesy. Podobně jako proces je i každé vlákno v systému jednoznačně rozlišeno pomocí identifikátoru, který nese název TID (thread

identifier). V praxi je TID přidělován ze stejného číselného prostoru jako PID. Hlavní vlákno sdílí svůj identifikátor s procesem a platí tedy, že TID odpovídá PID. Analogicky jako u procesu je možné i u vlákna zjistit jeho TID zavoláním funkce `gettid()`. [11]

3.3 Životní cyklus procesu

Během svého života proces prochází několika stavy. Život začíná jeho vytvořením a končí v okamžiku, kdy po něm jeho rodičovský proces uklidí. Stavy procesu jsou tyto: [11]

- **běžící** – Procesor vykonává programový kód procesu
- **zastavený** – Proces obdržel některý ze signálů `SIGSTOP`, `SIGTTIN`, `SIGTTOU` nebo `SIGTSTP` a byl tím zastaven. Žádný procesor nevykonává kód tohoto procesu.
- **pozastavený** – Provádění kódu bylo přerušeno signálem `SIGTRAP` kvůli splnění podmínky (například kvůli debuggeru). Tento stav je podobný předchozímu stavu.
- **zombie** – Provádění kódu bylo dokončeno, a proces již neběží na žádném procesoru. Proces v tomto stavu setrvá do okamžiku, než jeho rodičovský proces požádá jádro o jeho odstranění.

4 PROJEKTY PRO OS LINUX UMOŽŇUJÍCÍ VYTVOŘIT CLUSTER

Následující přehled nepopisuje zdaleka všechny dostupné projekty, ale je zaměřen pouze na ty z nich, které jsou ve vývojovém stádiu umožňujícím nasazení v běžném provozu. Při výběru hraje roli provozování aplikací třetích stran jako Mathematica, Matlab a podobně. Je tedy nutné zaměřit se na projekty umožňující vytvářet cluster, který má povahu SMP nebo NUMA systému.[7],[12]

4.1 Beowulf

Projekt Beowulf vznikl v letech 1993 a 1994 jako výsledek vývoje paralelního počítače zkonstruovaného z běžně dostupných komponent s využitím volně šiřitelného software. Vývoj tohoto projektu probíhal v NASA na pracovišti CESDIS. V tehdejší době si nedostupnost síťových přepínačů vyžádala poměrně rozsáhlé úpravy síťových ovladačů jádra Linux. Z hlediska hardware nebylo nutné provádět žádné úpravy. Tento projekt není vázán čistě na OS Linux, ale je možné jej provozovat i na jiných opensource uniových systémech (např. BSD, Solaris, …).

Beowulf je vhodný pro výstavbu superpočítačů zejména pro náročné výpočetní úlohy. Využívá MPP přístup a jsou podporovány jazyky C, C++ a Fortran. Nevýhodou projektu je, že ho nelze použít pro provozování běžných aplikací nebo aplikací třetích stran. Projekt je v přehledu zmíněn zejména pro jeho velkou popularitu v oblasti opensource cluster projektů. [8]

Více informací o projektu je dostupných v [9].

4.2 OpenMosix

Jedná se o komplexní sadu clusterovacích nástrojů pro OS Linux vycházející z projektu Mosix, které umožňují vytvářet high performance cluster. Tento projekt rozkládá zátěž na principu migrace jednotlivých procesů (v okamžiku, kdy proces volá funkci fork()) mezi uzly clusteru, a cluster se pak navenek tváří jako homogenní SMP stroj. Takovýto přístup si vyžádal mimo jiné úpravy plánovače jádra Linux. OpenMosix poskytuje nejen upravené jádro Linux, respektive jeho zdrojové kódy, ale i sadu nástrojů pro správu a sledování provozu clusteru.

Tento produkt je vhodný pro paralelní aplikace nebo aplikace pracující s velkým objemem dat. Nespornou výhodou je dostupnost širokého množství více či méně specializovaných liveCD distribucí využívajících OpenMosix, které umožňují zájemci vystavět cluster rychle a bez dlouhého zkoumání problematiky. Nevýhodou je oficiálně ukončený vývoj stabilní větve a stagnující vývoj varianty s podporou 64 bitových procesorů a Linuxu verze 2.6. Absence podpory Linux verze 2.6 se sebou nese rovněž absenci podpory vláken, jejíž kvalitní implementace je k dispozici právě od verze 2.6. [13]

Více informací o projektu je dostupných v [14] .

4.3 OpenSSI

OpenSSI je projekt zaměřený na high performance computing a high availability clustering. Pro rozložení zátěže využívá rovněž principu migrace jednotlivých procesů mezi uzly a tím vytváří homogenní SMP prostředí. Další z užitečných vlastností je sdílení paměti a obsahu adresáře `/dev` v rámci celého clusteru, to znamená, že procesy mohou sdílet paměť a hardware všech počítačů v clusteru. Velkou výhodou je možnost přidávání a odebírání jednotlivých počítačů za běhu.

Podobně jako v případě OpenMosix existuje i v případě OpenSSI liveCD, které pomáhá urychlit budování clusteru.

V současné době projekt nepodporuje práci s vlákny, což je poměrně velký deficit. [15]

Více informací o projektu je dostupných v [16].

4.4 Kerrighed

Projekt Kerrigherd je rovněž zaměřený na high performance computing a high availability clustering. Kerrighed je implementován jako rozšíření jádra Linux, sada modulů jádra, nástrojů pro práci s clusterem a v neposlední řadě knihoven umožňujících začlenit komunikaci s clusterem do vytvářené aplikace. Pro rozložení zátěže používá techniku migrace nejen procesů, ale i jednotlivých vláken (funkce migrace vláken je vývojáři projektu v současné době deaktivována, nicméně ve starších verzích je aktivní), což umožňuje provozovat velmi širokou škálu aplikací. Dalšími dostupnými vlastnostmi je možnost sdílení veškeré dostupné paměti v rámci celého clusteru, nebo možnost přidávání a odebírání jednotlivých počítačů do clusteru za běhu. Kerrighed je poměrně mladý

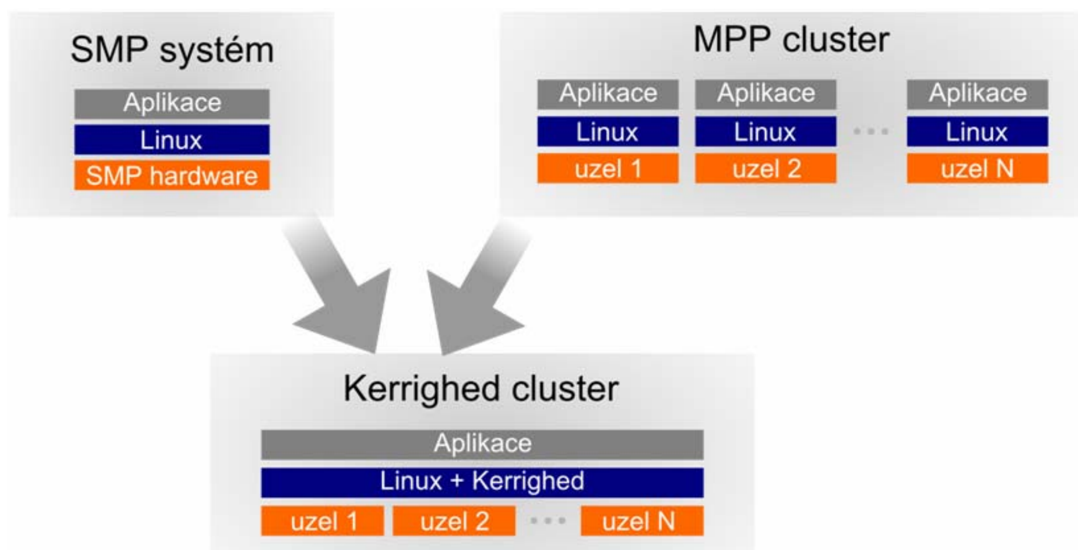
dynamicky se vyvíjející projekt, který obsahuje podporu pro vícejádrové, víceprocesorové a 64 bitové systémy. V současné době je dostupné i demonstrační liveCD.

Jedinou nevýhodou je provázanost Kerrighed na danou verzi jádra Linux, což je v současné době 2.6.20.

Více informací o projektu je dostupných v [17].

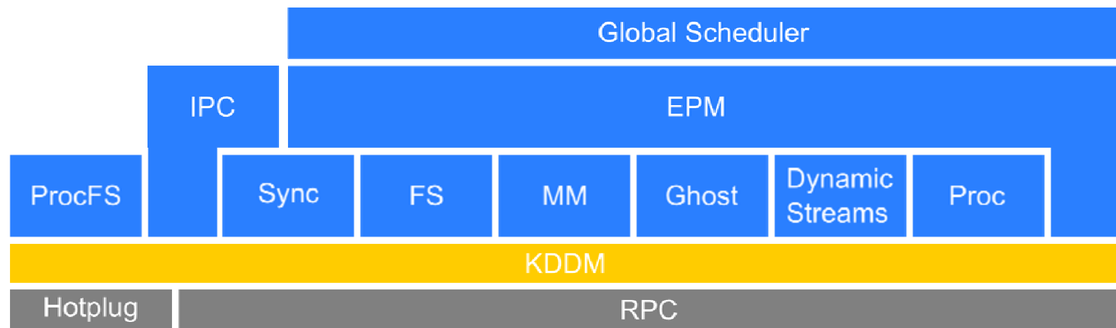
5 ARCHITEKTURA KERRIGHED

Kerrighed v sobě kombinuje prvky počítačů architektury SMP a clusteru. Nad clusterem běžných počítačů vytváří virtuální SMP (NUMA) systém. Na blokovém schéma (*Obr. 3*) je znázorněno, jaké jsou rozdíly mezi SMP, Kerrighed a MPP clusterem a naopak, které vlastnosti obou řešení Kerrighed kombinuje. V případě hardwarové implementace SMP systému jde o drahé řešení (vysoká cena SMP hardware, která roste s počtem instalovaných procesorů), v případě MPP clusteru je možné využít větší množství běžných počítačů (stanic), avšak je nutné vyřešit rozdělení provozované aplikace na N nezávislých celků. Kerrighed v sobě kombinuje možnost provozování nemodifikovaných aplikací na běžných počítačích.



Obr. 3: Rozdíl mezi tradičním SMP systémem,
MPP clusterem a Kerrighed clusterem

Projekt Kerrighed se skládá celkem ze 13 částí, jejichž struktura je znázorněna na blokovém schéma (*Obr. 4*). Moduly nejnižší vrstvy patří ke komunikačnímu subsystému Kerrighed zajišťujícímu předávání informací mezi stanicemi clusteru. Prostřední vrstva implementuje mechanismy pro základní distribuované služby. Moduly vyšších vrstev poskytují služby využívané aplikacemi a uživateli.



Obr. 4: Blokové schéma architektury Kerrgihed

Kerrighed obsahuje následující moduly: [19]

- **Capability** – jde o rozhraní, pomocí něhož je možné provádět jemné korekce chování jednotlivých procesů. Je využito k aktivaci a deaktivaci některých funkcí Kerrighed přímo na úrovni samostatných procesů. Jde spíše o rozhraní jednotlivých modulů, proto není zakresleno v blokovém schéma.
- **RPC (Remote Procedure Calling)** – implementuje nejvyšší úroveň správy distribuovaných služeb. Nabízí funkce blízké těm, které jsou obsaženy v RPC mechanismech. Spoustu explicitní komunikace pak obstarává toto rozhraní.
- **Hot Plug** – tento modul je zodpovědný za sledování jednotlivých stanic, jejich přidávání, odebírání a případná selhání. Zároveň je využíván ke správě clusteru a jeho konfiguraci.
- **KDDM (Kerrighed Distributed Data Manager)** – jde o mechanismus, pro distribuovanou správu dat. Nabízí nástroje nejvyšší úrovně pro efektivní a rychlý přístup k datům uloženým na vzdálených úložištích.
- **ProcFS (Proc FileSystem)** – implementuje globální systém souborů /proc. Poskytuje stejnou adresářovou strukturu a stejné soubory jako /proc na běžném systému s tím rozdílem, že zde jsou dostupná data pro celý cluster (je tedy možné sledovat globální obsazení paměti, všechny spuštěné procesy atd.).
- **Sync (Synchronisation)** – implementuje základní distribuované synchronizační mechanismy jako zámky, semafory a podmíněné proměnné. Ke své implementaci využívá synchronizační mechanismy na uživatelské úrovni (IPC (Inter-Process Communication) semafory, Posix vláknovou synchronizaci, atd.)

- **IPC** – tento modul implementuje distribuovanou verzi IPC mechanismu. Závisí na MM modulu pro sdílení segmentů paměti a na semaforech obsažených v Sync modulu.
- **FS (FileSystem)** – implementuje podporu pro migraci otevřených souborů a sdílení ukazatele na soubor pro procesy spuštěné na jiné stanici. Ve vývoji je podpora pro správu souborů v rámci clusteru.
- **MM (Memory Management)** – tento modul je zodpovědný za migraci adresního prostoru procesů a sdílení paměti dostupné globálně v rámci clusteru.
- **Dynamic Streams** – implementuje základní mechanismus použitý pro efektivní migraci otevřených datových proudů jako roury, sockety, znaková zařízení atd.
- **Ghost** – implementuje základní vrstvu použitou pro export a import metadat procesů. Je využit pro migraci, kontrolu a duplikaci procesů.
- **Proc** – implementuje distribuovanou správu procesů. Je zodpovědný za globální pojmenování procesů, vzdálenou signalizaci apod.
- **EPM (Enhanced Process Management)** – modul provádí migraci procesů, kontrolu, vzdálené založení procesu a distribuovanou správu vláken.
- **Global Scheduler** – globální plánovač Kerrighed, který implementuje rozdílné globální politiky plánování procesů.

II. PRAKTICKÁ ČÁST

6 NASAZENÍ CLUSTERU

Nasazení clusteru sestaveného z několika běžně dostupných počítačů je velmi zajímavou alternativou k nákupu jednoúčelového superpočítače. Cluster lze sestavit z běžně dostupných počítačů a běžně dostupného síťového vybavení. Je možné tedy využít vyřazené počítače nebo například využít počítače v nočních hodinách nebo o víkendu, kdy nejsou využívány ke svému primárnímu účelu. Provoz takového clusteru se sebou nenese nutnost investic do žádného dalšího hardware.

Na UTB ve Zlíně by takový cluster bylo možné využít k provozu časově náročných algoritmů nebo k výuce programování paralelních aplikací.

Při nasazení výpočetního clusteru se počítá s využitím počítačů ve studovnách v době, kdy nejsou využívány studenty. To zajišťuje dostupnost velkého množství počítačů byť po omezenou dobu. Navíc tyto počítače jsou pravidelně udržovány a modernizovány, a není nutné investovat ze stejných důvodů do počítačů dedikovaných pro cluster. Vzhledem k tomu, že počítačů je velké množství, by instalace operačního systému využitého pro cluster na jejich pevné disky byla velmi komplikovaná. Proto byla zvolena varianta zavádění operačního systému ze sítě.

6.1 Volba cluster projektu a distribuce

Pro praktické nasazení clusteru pro provoz na UTB bylo nutné vybrat vhodný projekt pro provoz clusteru a následně v závislosti na daném projektu bylo možné vybrat vyhovující distribuci operačního systému Linux.

Při výběru vhodného řešení clusteru byly stanoveny následující požadavky:

- aktivní vývoj a podpora aktuálních verzí jádra Linux
- provoz aplikací třetích stran jako Matlab, Mathematica, Blender
- možnost provozování na pracovních stanicích ve studovnách například v nočních hodinách bez zásahů do instalovaného software
- centralizovaná správa clusteru
- cenová dostupnost

Dané požadavky nejlépe splňuje projekt Kerrighed.

Při výběru vhodné distribuce je tedy vhodné zohlednit, že instalace Kerrighed vyžaduje kompilaci jádra Linux a úpravy startovacích skriptů.

Na distribuci jsou kladeny následující požadavky:

- možnost instalace aplikací třetích stran
- možnost kompilace jádra
- repositáře obsahující stabilní otestovaný software

Dané požadavky nejlépe splňuje distribuce Debian GNU/Linux ve verzi stable.

6.2 Technické řešení

Pro maximální zjednodušení při praktickém nasazení clusteru bylo přistoupeno k řešení, kdy operační systém jednotlivých stanic clusteru není instalován na jejich pevném disku, ale na boot serveru a stanice zavádí svůj operační systém ze sítě. Zároveň jakákoliv správa jednotlivých stanic je prováděna jen jednou, pouze na boot serveru, a změny se projeví okamžitě na všech stanicích. Tím byly splněny podmínky centrální správy a minimalizace zásahu do stávajícího software na stanicích.

Start systému je na straně stanic zajištěn pomocí standardu PXE (Preboot eXecution Environment), který ke své funkci na straně boot serveru vyžaduje funkční DHCP (Dynamic Host Configuration Protocol) a TFTP (Trivial File Transfer Protocol) server, pro další provoz je vyžadován NFS (Network File System) server. Služby DHCP a TFTP jsou využity pouze pro první fázi startu operačního systému – tedy získání IP adresy a stažení obrazu jádra a jeho spuštění. Pro další provoz operačního systému na stanicích je kořenový svazek připojen z boot serveru pomocí NFS. Jednotlivé role mohou být zastoupeny různými servery a nemusí být nutně provozovány na společném serveru. Je proto možné pro některé role využít existující infrastrukturu, pochopitelně podporující potřebné funkce.

Provedení sítě je věnována příloha P I.

6.3 Použitý hardware

Cluster se skládá ze 3 částí, boot serveru, síťové infrastruktury a jednotlivých stanic. Pro boot server byl využit počítač vybavený procesorem Intel Pentium 4 1,8GHz, osazený

256 MiB RAM a obsahující síťový adaptér o rychlosti 10/100 Mb/s. Pro účely instalace operačního systému a uložení dat byl využit 80 GB IDE pevný disk. Jako základ pro síťovou infrastrukturu byl využit 8-portový hub o rychlosti 10 Mb/s, který disponoval možností sledovat aktuální zatížení. Další částí síťové infrastruktury je kabeláž odpovídající rychlosti použitých switchů a síťových adaptérů. Pro stanice clusteru byly využity počítače vybavené procesorem Intel Pentium 4 1.8GHz, osazené 256 MiB RAM a obsahující rovněž síťový adaptér o rychlosti 10/100 Mb/s. Pro účely zavedení operačního systému ze sítě byly počítače vybaveny CD-ROM mechanikami, které byly využity pro zavedení PXE ze spustitelného CD.

Návody na instalaci a konfiguraci boot serveru a vytvoření systému pro stanice jsou součástí příloh PII a PIII.

6.4 Použití

Po instalaci potřebného software je nutné nastavit stanice tak, aby byly schopny zavést operační systém ze sítě za pomocí PXE. Nastavení zavádění systému z PXE je obvykle dostupné v BIOS (Basic Input-Output System) pod názvem **Boot from LAN**.

Po zapnutí stanice proběhne POST (Power On Self Test) a následně se BIOS stanice pokusí zavést systém z PXE. V případě úspěšného nalezení DHCP serveru jsou stanici přiděleny mimo jiné parametry jako IP adresa, maska sítě a adresa TFTP serveru, odkud má PXE stáhnout použitý zavaděč. Jako zavaděč slouží pxelinux, který na základě konfigurace zajistí stažení obrazu jádra a jeho spuštění s nastavenými parametry, mezi které patří rovněž adresa NFS svazku, který má být připojen jako kořenový adresář.

Po spuštění jádra Linux je provedena detekce hardware a připojení kořenového adresáře z NFS serveru. Po připojení kořenového adresáře je start systému řízen pomocí startovacích skriptů a konfiguračních souborů, které jsou uloženy právě na NFS serveru.

Při startu systému jsou rovněž zavedeny jaderné moduly (ovladače) Kerrighed (`/lib/modules/2.6.20-krg/extra/kerrighed.ko`) a dojde ke spuštění služeb poskytovaných modulem. Mezi nejdůležitější patří služba pro komunikaci mezi ostatními stanicemi, na kterých je rovněž zaveden modul Kerrighed. Tato služba zajistí, že všechny stanice mají informace o ostatních stanicích, tzn. identifikátor sezení, příslušnost k síti a unikátní identifikátor stanice (tzv. node ID).

Po inicializaci Kerrighed je spuštění systému dokončeno a je možné se k němu přihlásit. Veškerá práce s clusterem pak probíhá prostřednictvím přihlášení ke kterékoliv stanici, a to i vzdáleně pomocí ssh.

Po spuštění všech stanic jde vlastně jen o síť jednoprocesorových počítačů a vlastní cluster není inicializován. Správa clusteru nebo jeho jednotlivých stanic je prováděna prostřednictvím příkazu `krgadm`. Pomocí něj je možné celý cluster inicializovat, restartovat nebo vypnout, dále je možné jednotlivé uzly zařazovat do clusteru nebo je z něj vyřazovat rovnou za jízdy (tzv. hotplug). Inicializace clusteru se provádí v okamžiku, kdy jsou všechny stanice spuštěny příkazem `krgadm cluster start`. O úspěchu inicializace je možné se přesvědčit z výpisu příkazu `cat /proc/cpuinfo`, který vypíše všechny dostupné procesory a jejich vlastnosti.

Funkce clusteru jsou ovládány prostřednictvím tzv. capabilities – schopností, které jsou nastavovány pomocí příkazu `krgcapset`. Schopnosti mohou být nastaveny globálně nebo mohou být definovány explicitně pro jednotlivé procesy a jsou definovány ve 4 sadách:

- **Efektivní (effective) -** schopnosti aplikované jádrem operačního systému na procesy.
- **Povolené (pertmitted) -** schopnosti, které může proces převzít (omezuje nadmnožinu efektivních, dědičných a dědičných povolených sad). Pokud proces některé schopnosti ztratí, nemůže je už znovu nabýt
- **Dědičné efektivní (inheritable effective) -** schopnosti stejného typu jako efektivní, aplikované na nový proces při volání `execve()`, které má na starost spuštění programu pomocí ukazatele na jeho spustitelný soubor.
- **Dědičné povolené (inherirable permitted)** - schopnosti stejného typu jako povolené, aplikované na nový proces při volání funkce `execve()`.

V současné době (ve verzi 2.3.0) jsou implementovány následující schopnosti:

- **CHANGE_KERRIGHED_CAP -** povoluje změnu schopností.
- **CAN_MIGRATE -** povoluje procesu migraci mezi stanicemi clusteru.

- **DISTANT_FORK -** tato schopnost je využita voláním `fork()` pro určení, zda může být nový proces spuštěn na jiné stanici. Úspěšnost této operace není zaručena.
- **SEE_LOCAL_PROC_STAT -** tato schopnost povoluje procesu vidět obsah adresáře `/proc` příslušející pouze ke konkrétní stanici na rozdíl od globalizované varianty platné pro celý cluster.

Po inicializaci je ve výchozím stavu cluster nastaven tak, že migrace procesů není povolena, a tudíž cluster není možné efektivně využít. Pro povolení migrace procesů je nutné přidat schopnost **CAN_MIGRATE** do dědičné efektivní (inheritable effective) skupiny pomocí příkazu `krgcapset -d +CAN_MIGRATE`. Cluster s takto nastavenými schopnostmi začne vyrovnávat zátěž tak, že nově spuštěné procesy migruje na méně zatížené stanice.

Migraci jednotlivých procesů mezi stanicemi lze velmi dobře sledovat pomocí nástroje **htop** (spustitelného stejnojmenným příkazem). Pro otestování funkcí clusteru je součástí Kerrighed v adresáři `/usr/src/kerrighed-2.3.0/tests` sada testů. Na snímcích obrazovky (Obr. 5 a Obr. 6) je zachycen výstup programu htop během spuštěného testu `fork_test` z podadresáře `proc`. Podstata tohoto testu spočívá ve vytvoření mnoha stejných procesů vykonávajících jednoduchý výpočet.

```
195.113.98.182 :
Soubor  Úpravy  Pohled  Rolování  Záložky  Nastavení  Nápověda

  1  [|||||||||||||||||||||||||||||||||||||||||||||100.0%]     Tasks: 45 total, 3 running
  2  [                                              0.0%]     Load average: 1.20 0.63 0.43
  3  [||                                            2.0%]     Uptime: 16 days, 19:23:08
  4  [|                                             1.0%]
  Mem[||||||||||||||||||||||||||||||          402/992MB]
  Swp[                                            0/0MB]

  PID USER     PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
175025 root     19   0  1664   208   128 S 10.4  0.0  0:01.01 ./fork_test
175032 root     24   0  1664   124    44 R  9.6  0.0  0:00.52 ./fork_test
175027 root     21   0  1664   188   108 S  9.6  0.0  0:00.78 ./fork_test
175039 root     25   0  1664   124    44 R  8.8  0.0  0:00.21 ./fork_test
174806 root     15   0  2284  1164   924 R  0.8  0.1  0:16.65 htop
175024 root     15   0  1664   376   300 S  0.0  0.0  0:01.03 ./fork_test
174807 root     15   0  7992  2616  2164 S  0.0  0.3  0:00.19 sshd: root [priv]
174798 root     15   0  7996  2612  2164 S  0.0  0.3  0:00.09 sshd: root@pts/1
174811 root     15   0  2772  1544  1228 S  0.0  0.2  0:00.04 -bash
    1 root     15   0  1980   688   592 S  0.0  0.1  0:00.34 init [2]
164900 root     15   0  1624   164   100 S  0.0  0.0  0:00.09 logsave -s /var/log/fsck/checkfs fsck -C -R -A -a
165102 root     15   0  5276  1044   688 S  0.0  0.1  0:00.01 /usr/sbin/sshd
165168 root     16   0  1648   512   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty1
165169 root     16   0  1644   516   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty2
165170 root     16   0  1644   516   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty3
165171 root     16   0  1648   520   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty4
165172 root     15   0  1644   512   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty5
165174 root     15   0  1648   520   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty6
174784 root     15   0  7996  2612  2164 S  0.0  0.3  0:00.08 sshd: root@pts/0
174788 root     16   0  2764  1500  1192 S  0.0  0.1  0:00.01 -bash
174802 root     15   0  2768  1496  1188 S  0.0  0.1  0:00.01 -bash
230435 root     15   0  1624   164   100 S  0.0  0.0  0:00.07 logsave -s /var/log/fsck/checkfs fsck -C -R -A -a
230631 root     15   0  5276  1048   688 S  0.0  0.1  0:00.00 /usr/sbin/sshd
230697 root     15   0  1648   520   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty1
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit

195.113.98.182 :    195.113.98.182 :
```

Obr. 5: Snímek obrazovky htop zobrazující test bez povolené migrace procesů

```
195.113.98.182 :
Soubor  Úpravy  Pohled  Rolování  Záložky  Nastavení  Nápověda

  1  [|||||||||||||||||||||||||||||||||||||||||||||100.0%]     Tasks: 49 total, 7 running
  2  [|                                             1.0%]     Load average: 2.37 0.75 0.37
  3  [|||||||||||||||||||||||||                    46.0%]     Uptime: 16 days, 19:12:33
  4  [||||||||||||||||||||                         36.5%]
  Mem[||||||||||||||||||||||||||||||          402/992MB]
  Swp[                                            0/0MB]

  PID USER     PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
174974 root     25   0  1660   120    44 R 100.  0.0  0:00.04 ./fork_test
174965 root     20   0  1660   164     0 S 30.0  0.0  0:00.95 ./fork_test
174966 root     22   0  1660   120    44 R 18.6  0.0  0:00.78 ./fork_test
174968 root     22   0  1660   120    44 R 17.3  0.0  0:00.48 ./fork_test
174970 root     24   0  1660   120    44 R 12.0  0.0  0:00.41 ./fork_test
174972 root     24   0  1660   120    44 R  8.0  0.0  0:00.22 ./fork_test
174963 root     15   0  1660   372   300 S  4.7  0.0  0:01.18 ./fork_test
174806 root     16   0  2284  1164   924 R  0.7  0.1  0:10.88 htop
174964 root     15   0  1660   164     0 S  0.0  0.0  0:01.07 ./fork_test
174807 root     15   0  7992  2616  2164 S  0.0  0.3  0:00.18 sshd: root [priv]
174798 root     15   0  7996  2612  2164 S  0.0  0.3  0:00.09 sshd: root@pts/1
174811 root     15   0  2772  1544  1228 S  0.0  0.2  0:00.04 -bash
    1 root     15   0  1980   688   592 S  0.0  0.1  0:00.34 init [2]
164900 root     15   0  1624   164   100 S  0.0  0.0  0:00.09 logsave -s /var/log/fsck/checkfs fsck -C -R -A -a
165102 root     15   0  5276  1044   688 S  0.0  0.1  0:00.01 /usr/sbin/sshd
165168 root     16   0  1648   512   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty1
165169 root     16   0  1644   516   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty2
165170 root     16   0  1644   516   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty3
165171 root     16   0  1648   520   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty4
165172 root     15   0  1644   512   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty5
165174 root     15   0  1648   520   448 S  0.0  0.1  0:00.00 /sbin/getty 38400 tty6
174784 root     15   0  7996  2612  2164 S  0.0  0.3  0:00.08 sshd: root@pts/0
174788 root     16   0  2764  1500  1192 S  0.0  0.1  0:00.01 -bash
174802 root     15   0  2768  1496  1188 S  0.0  0.1  0:00.01 -bash
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit

195.113.98.182 :    195.113.98.182 :
```

Obr. 6: Snímek obrazovky htop zobrazující test s povolenou migrací procesů

Rozdíl mezi situací bez povolené migrace (Obr. 5) a situací s povolenou migrací (Obr. 6) je nejvíc patrný v levé horní části, kdy v prvním případě je zatížen všemi vytvořenými procesy pouze jeden procesor, zatímco v druhém případě jsou zatíženy i ostatní procesory.

V přílohách PV, PVII a PIX jsou k dispozici lokalizované manuálové stránky, které popisují veškeré možnosti použitých příkazů pro správu Kerrighed clusteru. Manuálové stránky v přílohách jsou lokalizovány ze zdroje [20].

7 PROGRAMOVÁNÍ PARALELNÍCH APLIKACÍ

Paralelní aplikace je program rozdělený na několik entit, které mohou být prováděny souběžně. Hlavní motivací pro rozdělení je, aby každá z entit mohla být provozována na samostatném výpočetním jádru, a tím byla zkrácena doba vykonávání programu. Dalším důvodem pro vytvoření takového programu je, že například grafické rozhraní aplikace je reprezentováno jednou entitou a výkonná část druhou. V praxi pak grafické rozhraní takové aplikace při vykonávání požadované akce dále reaguje na uživatelské vstupy, nebo poskytuje například informace o stavu akce. Proto, aby doba běhu paralelního programu byla kratší, je nutné, aby data jednotlivých entit byla co nejvíce nezávislá. V případě, že vykonání některé z entit je závislé na výsledku jiné, souběžně vykonávané entity, dochází ke zbytečnému zpoždění a efektivita aplikace klesá. V ideálním případě nedochází k žádnému zpoždění. V praxi však ne všechny problémy budou ideálně rozdělitelné na více entit a v některých případech doposud není znám způsob rozložení. Typickým případem rozložitelného problému může být sečtení dvou funkčních hodnot, kdy funkční hodnoty mohou být vypočteny zároveň a výsledné hodnoty jsou pak sečteny. Naproti tomu o obtížně rozložitelný problém jde v případě, kdy budou funkce vnořeny, protože výpočet vnější funkce bude možné zahájit až v okamžiku, kdy bude znám výsledek vnořené funkce.

V operačním systému Linux jsou pro programování paralelních aplikací dostupné dva typy entit, jsou jimi proces a vlákno. Zásadní rozdíl z hlediska programu je ten, že v případě vláken na rozdíl od procesů spolu vlákna sdílí alokovanou paměť a platí, že více vláken pracuje uvnitř jednoho procesu. [11]

7.1 Víceprocesová aplikace

Nejjednodušším způsobem realizace paralelní aplikace je víceprocesová aplikace. Víceprocesová aplikace funguje tak, že nejprve je spuštěn program, který je reprezentován svým procesem. Tento již běžící proces zajistí pomocí volání `fork()` vytvoření své kopie. Rodičem nově vytvořené aplikace je tedy původní proces. Nově vytvořený proces dědí ze svého rodiče pouze souborové deskriptory. S tímto faktem je nutné počítat a ošetřit souběh událostí v kritické sekci (tzv. Race Condition). Každý proces má dva druhy oprávnění, reálná a efektivní. Reálná oprávnění jsou oprávnění odpovídající uživateli, který program spustil a efektivní oprávnění v případě nastavení SUID příznaku odpovídají

vlastníku souboru s programem. Při vytváření nového procesu jsou tyto vlastnosti děděny a je nutné s tímto faktem rovněž počítat, protože představuje vysoké bezpečnostní riziko.

V okamžiku skončení procesu (skončení vykonávání programového kódu) je nastavena jeho návratová hodnota, podle které může rodičovský proces například kontrolovat úspěšnost běhu.

Ke zvláštní situaci dojde v okamžiku, kdy rodičovský proces skončí dříve, než proces jím vytvořený. V takovém případě je procesu přidělen jako rodič proces init s PID=1. Tento postup byl zvolen proto, že proces init čeká na skončení všech procesů a v případě jejich skončení zajistí korektní a hladké ukončení.

Následující příklad ukazuje nejjednodušší možnost vytvoření procesu. Všechny příklady v této kapitole lze přeložit pomocí překladače GCC verze 3.4 a 4.3.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
        int pid=fork();

        if (pid == -1) {
          /* chyba, proces nelze vytvorit*/
          }
        else if(pid != 0) {
          /* kod rodice */
          }
        else {
          /* kod potomka */
          }
        return 0;
}
```

V proměnné `pid` je uchována návratová hodnota při volání `fork()`, která vrací jednu ze tří hodnot. Při vytvoření nového procesu oba procesy pokračují od místa, kdy bylo vytvoření iniciováno. V případě nevytvoření nového procesu je vrácena hodnota -1. Pokud bylo vytvoření procesu úspěšné, je nově vytvořenému procesu vrácena hodnota 0, a rodičovskému procesu je vrácena hodnota PID nově vytvořeného procesu. V tomto

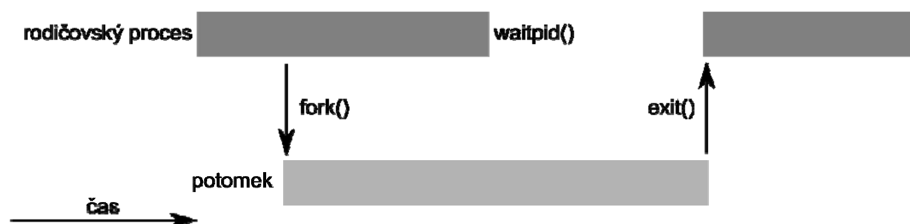
příkladě rodičovský proces může skončit dříve než jeho potomek a není kontrolována návratová hodnota.

Následující příklad ukazuje vytvoření procesu, čekání na něj a zjištění návratové hodnoty.

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>


int main() {

        int pid=fork();


        if (pid == -1) {

          /* chyba, proces nelze vytvorit*/

          }

        else if(pid != 0) {

          int stav;

          if (waitpid(pid, &stav, 0) == pid)

            if (WIFEXITED(stav)) {

            printf ("exited, %i\n",WEXITSTATUS(stav));

            }

          /* kod rodice */

          }

        else {

          /* kod potomka */

          }

        return 0;

}
```

Od předchozího příkladu se tento příklad liší v části kódu v oblasti, která bude vykonána v rodičovském procesu. Základem je funkce `waitpid()`, která do proměnné `stav` nastaví výsledek běhu procesu s PID předaným prostřednictvím proměnné `pid`. V případě úspěchu vrací PID procesu, na který rodičovský proces čekal. Na výsledek v proměnné

stav lze aplikovat řadu maker, která poskytují informace o tom, jak byl proces ukončen. V příkladu je využito makro WIFEXITED, které zjišťuje, zda proces skončil přirozeným způsobem a WEXITSTATUS, které vrací návratovou hodnotu procesu. Funkce takového programu je naznačena na obrázku (*Obr. 7*).



Obr. 7: Rodičovský proces čekající na potomka

Kromě funkce `waitpid()` jsou k dispozici ještě funkce `wait()` a `waitid()`. Funkce `wait()` čeká na libovolný podřízený proces, do proměnné stav ukončení, vrací jeho PID, funguje pouze v blokujícím režimu a neumožňuje sledovat konkrétní proces. Funkce `waitid()` má velmi podobné vlastnosti jako funkce `waitpid()`, liší se v použitém rozhraní pro předávání parametrů a zjišťování stavu procesu. Dále navíc obsahuje možnost explicitně definovat stavy procesu, na které bude čekat.

V předcházejících příkladech nebyla doposud zmíněna možnost komunikace mezi procesy. Protože procesy spolu nesdílí žádné proměnné, je velmi problematické, aby si procesy mezi sebou předávaly data, jako například výsledek výpočtu. Pro tyto účely lze využít funkce `pipe()`, která vytvoří tzv. rouru. Tento typ komunikace je možné použít pouze pro procesy, které mají stejného předchůdce a jde o poloviční duplex, který umožňuje posílat data pouze jedním směrem. Následující příklad využívá rouru k odeslání obsahu proměnné z rodičovského procesu do potomka, kde je sečtena s obsahem druhé proměnné a výsledek je vypsán.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
        int roura[2];
        pipe(roura);
```

```
	int pid=fork();

	if(pid != 0) { // rodic
		int a = 10;
		write(roura[1],&a,sizeof(a));
	} else { // potomek
		int a1, b=305;
		read(roura[0],&a1,sizeof(a1));
		close(roura[1]);
		close(roura[0]);
		printf("soucet cisel z roury je %d\n", a1 + b);
	}
	return 0;
}
```

Funkce `pipe()` vrací v poli o dvou prvcích souborové deskriptory, za jejichž pomoci lze přes rouru posílat data. V případě příkladu pro zápis do roury slouží deskriptor obsažený v `roura[1]` a pro čtení dat deskriptor uložený v `roura[0]`. Samotné čtení nebo zápis probíhá podobně jako v případě souboru uloženého na pevném disku. Po zápisu a přečtení požadovaných dat je vhodné rouru podobně jako soubor uzavřít.

Další možností meziprocesové komunikace jsou tzv. pojmenované roury, někdy též nazývané fifo, které je možné využít i pro komunikaci mezi procesy, které nemají stejného rodiče. Pojmenované roury jsou zvláštním typem souboru a proto práce s nimi je prakticky stejná, jako práce s obyčejným souborem. Podobně jako u souboru, je i v případě pojmenované roury nutné ji na začátku nejprve připojit pomocí příkazu `mkfifo()`, která jako parametry vyžaduje cestu k fifo souboru a režim přístupu podobně jako u funkce `open()`.

7.2 Vícevláknová aplikace

Další možností realizace paralelní aplikace je vícevláknový program. Práce s vlákny je z části velmi podobná práci s procesy a liší se zejména v tom, že vlákna využívají společnou paměť v rámci procesu, ve kterém byly vytvořeny. Vlastní práce s vlákny

probíhá pomocí knihovny libpthread, která má na starost implementaci rozhraní POSIX Threads a přidává některé funkce nad rámec normy POSIX. Knihovna je dostupná prostřednictvím hlavičkového souboru `pthread.h`, který musí být programem linkován. K vytvoření nového vlákna slouží funkce `pthread_create()`. Funkce má čtyři parametry, ukazatel na proměnnou typu pthread_t sloužící k identifikaci vláken, ukazatel na atributy vlákna, startovní funkci a obecný ukazatel na data. Atributy jsou nepovinné a lze vložit NULL. Pokud je vytvoření vlákna úspěšné, vrací funkce 0, pokud dojde k chybě, vrací funkce záporné číslo jako kód chyby. Nově vzniklé vlákno spouští startovací funkci a předává jí parametr zadaný při volání. Pro překlad zdrojových kódů v této kapitole pomocí překladače GCC je nutné program slinkovat s knihovnou pthread pomocí parametru –l (výsledný příkaz pro kompilaci pak je: `gcc zdrojovykod.c –l pthread`).

Následující příklad demonstruje program vytvoří dvě vlákna a demonstruje sdílenou paměť.

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>

int promenna;

void * zdroj (void *ukazatel)
    {
      printf("zdroj %s: spusten\n", (char *) ukazatel);
      promenna = rand();
      printf("vygenerovano cislo %d \n", promenna);
      sleep(5);
      printf("zdroj %s: ukoncen\n", (char *) ukazatel);
      return NULL;
    }

void * cil (void * ukazatel)
    {
      printf("cil %s: spusten\n", (char *) ukazatel);
      printf("zdroj vygeneroval cislo %d \n", promenna);
      printf("cil %s: ukoncen\n", (char *) ukazatel);
      return NULL;
    }
```

```
int main(){
	int retcode;
	pthread_t a,b;
	void * retval;
	retcode = pthread_create(&a, NULL, zdroj, "A");
	if (retcode != 0) fprintf(stderr, "create a failed %d\n", retcode);
	retcode = pthread_create(&b, NULL, cil, "B");
	if (retcode != 0) fprintf(stderr, "create b failed %d\n", retcode);
	retcode = pthread_join(a, &retval);
	if (retcode != 0) fprintf(stderr, "join a failed %d\n", retcode);
	retcode = pthread_join(b, &retval);
	if (retcode != 0) fprintf(stderr, "join b failed %d\n", retcode);
	return 0;
}
```

Startovací funkce, v tomto případě `zdroj` a `cil`, vrací hodnotu vypu `void` a lze ji využít ke vrácení výsledku, v tomto případě vrací pouze `NULL`. Startovací funkci je rovněž předávána hodnota ukazatele `ukazatel` získaného před vytvořením samotného vlákna.

Běh vlákna je ukončen několika způsoby podobně jako v případě funkce `main()` v procesu. Buď dojde k návratu hodnoty, nebo je možné kdekoliv ve vlákně zavolat funkci `pthread_exit()`, jejíž funkce je velmi podobná funkci `exit()` pro ukončení celého procesu. Speciálním případem je zrušení vlákna, které je možné provést zavoláním funkce `pthread_cancel()` na vlákno. Tato funkce ruší vlákno tak, že je zrušen běh vlákna. V opačném případě na vlákno čeká hlavní vlákno z důvodu synchronizace a převzetí návratové hodnoty. K čekání na vlákno slouží funkce `pthread_join()`, která obsahuje dva parametry. Jsou jimi identifikátor vlákna a adresa pro uložení návratové hodnoty. Toto čekání je blokující a může selhat v případě, že by mohlo způsobit uváznutí (tzv. deadlock). K funkci `pthread_join()` existuje i její neblokující ekvivalent `pthread_tryjoin_np()` a verze s časovým limitem `pthread_timejoin_np()`, které ale nejsou součástí specifikace POSIX a tudíž jsou nepřenositelné.

Existuje možnost na vlákno nečekat a vlákno převést do tzv. odloučeného stavu pomocí funkce `pthread_detach()`. Na odloučené vlákno už není možné zavolat `pthread_join()`, takové vlákno v okamžiku kdy skončí, vrátí všechny alokované prostředky. Vlákno, které je odloučené už nelze vzít zpět.

Jednou z důležitých obslužných funkcí je tzv. úklidové makra `pthread_cleanup_push()` a `pthread_cleanup_pop()`. Tyto makra jsou

zavolána v případě, že je vlákno zvenku zrušeno (například pomocí `pthread_cancel()`). Úklidová funkce je deklarována a zavolána podobně jako startovací funkce pro vlákno. Praktické použití úklidového makra v programu probíhá zásobníkovým způsobem, lze tedy uložit více volání a ty se vykonávají v opačném pořadí než byla vložena. Typické použití pro úklidová makra je uvolnění paměti alokované ve vlákně.

7.3 Synchronizační prostředky

Pro účely synchronizace událostí při vstupu do kritické sekce programu je k dispozici několik prostředků. Následující popis zmiňuje ty nejvýznamnější z nich.

7.3.1 Semafor

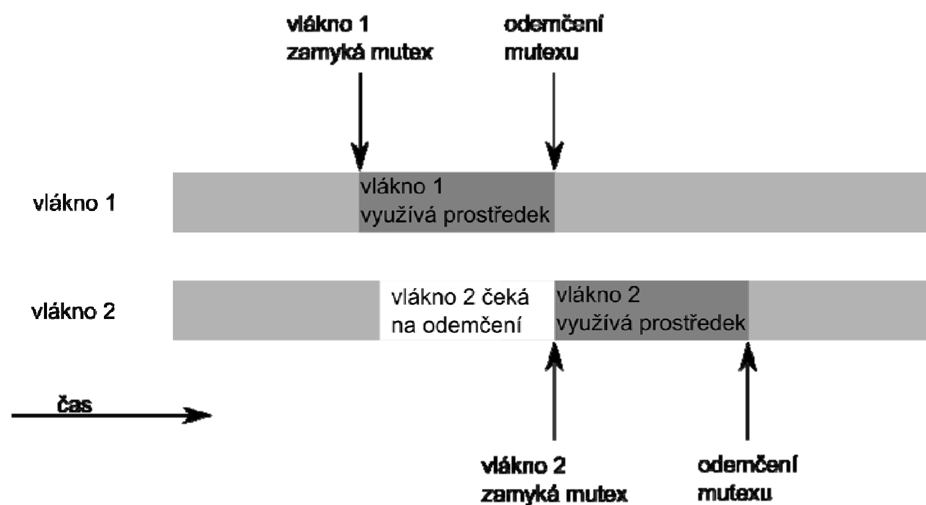
Semafor není obecně použitelný k přenosu dat mezi procesy. Jedná se o počitadlo určené k přidělování zdrojů. Pro korektní funkci semaforu je nutné zajistit atomičnost operací nad semafory z důvodu, aby mezi testem a změnou semaforu nemohl hodnotu semaforu změnit jiný proces. Nejjednodušším typem semaforu je tzv. binární semafor, který řídí přístup právě k jednomu objektu. V případě, kdy semafor může nabývat více hodnot funguje tak, že pokud je hodnota semaforu kladná, proces využije prostředek a sníží hodnotu semaforu o 1. V případě, že je hodnota semaforu 0 je proces uspán do doby, než jsou zdroje uvolněny a semafor nabude kladné hodnoty. [11],[21]

Pro práci se semafory je k dispozici několik funkcí:

- `sem_init()` slouží pro inicializaci semaforu.
- `sem_wait()` slouží pro vstup do kritické sekce. Pokud je sekce obsazena (hodnota semaforu je rovna 0), pak proces čeká na uvolnění sekce.
- `sem_trywait()` slouží pro vstup do kritické sekce. Je-li sekce obsazena, funkce vrací chybu EAGAIN.
- `sem_post()` slouží k ukončení kritické sekce.
- `sem_getvalue()` vrací hodnotu semaforu.
- `sem_destroy()` uvolňuje všechny zdroje spojené se semaforem.

7.3.2 Mutex

Zvláštním případem semaforu je tzv. mutex, který lze využít pro účely synchronizace vláken. Název je zkratkou mutual exception (vzájemné vyloučení). Princip funkce mutexu je takový, že do určité oblasti bude mít v daný okamžik přístup pouze jediné vlákno (které provedlo operaci zamčení mutexu). Ostatní vlákna musí do odemčení mutexu čekat, nebo mohou provádět jinou činnost, pro kterou není důležitý přístup do kritické sekce. Fungování mutexu popisuje obrázek (Obr. 8). [11],[21]



Obr. 8: Funkce mutexu

Pro práci s mutexy jsou k dispozici následující funkce:

- `pthread_mutex_init()` slouží k inicializaci mutexu.
- `pthread_mutex_lock()` slouží k zamčení kritické sekce pomocí mutexu.
- `pthread_mutex_unlock()` odemyká kritickou sekci.
- `pthread_mutex_destroy()` na konci vlákna nebo kritické sekce slouží ke zrušení mutexu.

7.4 Programování aplikace pro Kerrighed

Aplikace provozovaná na Kerrighed clusteru mohou mimo běžných volání využít i volání specifická pro Kerrighed. Popis volání poskytovaných knihovnou libkerrighed je dostupný v přílohách PVII, PX a PXI, které obsahují manuálové stránky použitelných volání. Manuálové stránky v přílohách jsou lokalizovány ze zdroje [20].

Proto, aby mohla být využita knihovna libkerrighed je nutné, aby ve zdrojovém kódu byl includován hlavičkový soubor `kerrighed.h`. Dostupná volání jsou `krgcapset()`, `migrate()` a `migrate_self()`. Volání `krgcapset()` umožňuje programu modifikovat své schopnosti využitelné Kerrighed. Schopnosti jsou nastavovány (jsou při volání zapsány, nelze zjistit stav) prostřednictvím dat předaných ve struktuře `krg_cap_t`, která obsahuje čtyři položky typu `int`. Pomocí každé položky je při zavolání nastavena příslušná sada schopností, a to tak, že jednotlivé bity reprezentují povolení nebo zakázání jednotlivých schopností. Nejnižší bit odpovídá schopnosti CHANGE_KERRIGHED_CAP, další odpovídá CAN_MIGRATE, následuje DISTANT_FORK a 9. bit odpovídá vlastnosti SEE_LOCAL_PROC_STAT. Ostatní schopnosti nejsou v současné době implementovány.

Pomocí volání `migrate()` je možné migrovat proces zadaného PID na stanici se zadaným nodeid. Volání `migrate_self()` je ekvivalent volání `migrate()`, u kterého není zadáváno PID a migrován je pouze proces, ze kterého je provedeno volání na stanici se zvoleným nodeid.

8 TEST

Teoretický surový výpočetní výkon clusteru při zapojení například čtyř stejně výkonných počítačů bude odpovídat čtyřnásobku výkonu jednoho z počítačů. Z toho se dá odvodit, že teoretická doba běhu výpočtu se zkrátí na jednu čtvrtinu původní doby při běhu pouze na jednom počítači. V reálném clusteru však určitý čas navíc zabere migrace procesů, komunikace stanic mezi sebou atd. Délka této doby, která není využita k výpočtu závisí hlavně na charakteru dané úlohy a v neposlední radě na programové realizaci spuštěné aplikace. Všechny testy byly prováděny na clusteru se čtyřmi stanicemi. Bohužel nebylo možné plně otestovat všechny paralelně implementované aplikace, protože v Kerrighed je vývojáři projektu v současné verzi podpora migrace vláken dočasně vyřazena z důvodu jejího přepracování.

8.1 Vlastní testovací aplikace

Pro účely testu byly vytvořeny dvě varianty víceprocesové testovací aplikace. Procesy testovací aplikace mají za úkol zatěžovat procesor, a tím otestovat reakci migračního mechanismu plánovače procesů. Pro porovnání výkonnostního nárůstu bylo provedeno měření doby běhu na jedné stanici a doba běhu na clusteru.

Následující zdrojový kód reprezentuje jednodušší variantu, která vytvoří dva procesy:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

double fibonacci(int n) { /*vraci Nte cislo fibbonacciho posloupnosti*/
    double predminuly=1,minuly=1,soucet=1;
    int k;
    for (k=1; k<n;k++) {
            soucet=predminuly+minuly;
            predminuly=minuly;
            minuly=soucet;
      }
    return soucet;
   }

int main() {
```

```
        int pid=fork();
        if (pid == -1) {
          printf ("chyba, proces nelze vytvorit");
          }
        else if(pid != 0) {
            /*rodic*/
            int i,k;
            for (k = 0; k<300; k++) {
                printf("R spusten blok %d\n",k);
              for (i = 0; i<32000; i++) {
                        fibonacci(40);
                };
            };
        /*konec rodice*/
          }
        else {
        /* potomek */
        int i,k;
          for (i = 0; i<300; i++) {
              printf ("P spusten blok %d\n",i);
              for (k = 0; k<32000; k++) {
                      fibonacci(40);
                };
            };
        /* konec potomka */
          }
        return 0;
}
```

Tento testovací program vytvoří dva procesy, které nezávisle na sobě v cyklech spouští funkci `fibonacci`. Testovací program byl přeložen překladačem GCC verze 3.4.6 pomocí příkazu `gcc test_procesy.c –o test_procesy`. Testování doby běhu na jedné stanici bylo provedeno s inicializovaným clusterem, avšak nebyla povolena migrace procesů (CAN_MIGRATE) ani vzdálené vytváření procesů (DISTANT_FORK). Měření doby běhu bylo provedeno pomocí příkazu `time test_procesy`. Pro otestování nárůstu výkonu při provozu na clusteru byly podle kombinace povolených schopností provedeny následující testy:

- pouze s povolenou migrací procesů (CAN_MIGRATE)
- pouze s povoleným vzdáleným vytvářením procesů (DISTANT_FORK)

- povolenou migrací a vzdáleným vytvářením procesů (CAN_MIGRATE, DISTANT_FORK)

Při testu, kdy byly oba procesy spuštěny na jedné stanici, bylo dosaženo času 18,5s. Během testu na clusteru byl při různých nastaveních schopností čas shodný, vždy 10,0s. Teoretický předpokládaný nárůst výkonu by měl být v tomto případě dvojnásobný, avšak praktickým měřením byl změřen nárůst výkonu 1,85.

Složitější varianta testovacího programu umožňuje jednoduchou úpravou vytvořit libovolné množství podřízených procesů.

Následující kód reprezentuje složitější variantu, která vytváří 8 podřízených procesů.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

double fibonacci(int n) /*vraci Nte cislo fibbonacciho posloupnosti*/
    {
      double predminuly=1,minuly=1,soucet=1;
      int k;
      for (k=1; k<n;k++) {
            soucet=predminuly+minuly;
            predminuly=minuly;
            minuly=soucet;
      }
      return soucet;
    }

int main()
{
  int pid,waitPid,i;
  int pocProc=8;    //zde je mozne volit, kolik procesu bude vytvoreno
  for (i=0; i < pocProc; i++) {
      pid = fork();
      if (pid == -1) pocProc=pocProc-1;
      if (pid==0) {
            i=pocProc;
            int a,b;
            for (a = 0; a<300; a++) {
          for (b = 0; b<32000; b++) {
```

```
                    fibonacci(40);
          };
            };

          }
      else
      {
         printf("rodic vytvoril potomka\n");
      }
  }
      int stav;
      if (pid!=0) {
    for (i=0; i < pocProc; i++) {
      waitPid = wait (&stav);
      printf("%d. potomek s PID = %d byl ukoncen\n",i,waitPid);
    }
  }

  return 0 ;
}
```

Tento program vytvoří v cyklu několik podřízených procesů, které v cyklech provádí funkci fibonacci. Testovací program byl rovněž přeložen překladačem GCC verze 3.4.6 pomocí příkazu `gcc procesy_volitelne.c –o procesy_volitelne`. Testování doby běhu na jedné stanici bylo provedeno za stejných podmínek jako v případě jednodušší varianty testu. Při testu, kdy bylo spuštěno 8 podřízených procesů na jednom procesoru, bylo dosaženo času 76,8 s. Protože budou zatíženy všechny stanice clusteru, bude teoretické maximální zrychlení čtyřnásobné. Při testu s povolenou schopností CAN_MIGRATE bylo dosaženo času 24,6s, což reprezentuje zrychlení 3,12. Při testech s povolenými schopnostmi DISTANT_FORK nebo DISTANT_FORK zároveň s CAN_MIGRATE bylo dosaženo času 20,2s, což reprezentuje zrychlení 3,8. V tomto případě se povolení schopnosti DISTANT_FORK projevilo pozitivně na době běhu testu, kdy nebylo nutné již spuštěný proces migrovat, ale bylo provedeno jeho spuštění na jiném uzlu, který měl k dispozici volné systémové prostředky.

8.2 Překlad programů

Jednou z možností nasazení clusteru může být například kompilace (překlad) rozsáhlého projektu, nebo například kompletní distribuce operačního systému. Velmi vhodné je,

pokud při překladu daného projektu je využito nástroje make. Při překladu pomocí make je možné zvolit, kolik procesů překladače má být zároveň spuštěno. Tato vlastnost je velmi užitečná, pokud má překlad probíhat na víceprocesorovém počítači nebo clusteru. Pro otestování rychlosti kompilace byl zvolen multimediální přehrávač MPplayer ve verzi 1.0rc2 dostupný z [22]. Archiv obsahující zdrojový kód byl rozbalen a v adresáři se zdrojovým kódem byla provedena konfigurace překladu pomocí příkazu: `./configure`.

Pro změření doby překladu na jednom počítači byl vypnut mechanismus migrace procesů a překlad byl proveden pomocí příkazu: `time make –j 4`. Pomocí toho příkazu bude doba překladu změřena programem time a parametr `–j` udává počet zároveň spuštěných procesů překladače.

Pro změření doby překladu na všech počítačích clusteru byl aktivován mechanismus migrace procesů a překlad byl proveden stejným příkazem jako v případě měření na jednom počítači.

Vzhledem k tomu, že v použité verzi Kerrighed je přítomna chyba, při níž nejsou s procesem korektně migrovány deskriptory otevřených souborů, nebylo možné test úspěšně dokončit. Empirickým pozorováním částí překladu však bylo zjištěno, že dochází ke znatelnému zrychlení.

9 ZÁVĚREČNÁ DOPORUČENÍ

9.1 Doporučení pro výběr hardware

Pro boot server je možné využít počítač osazený procesorem Pentium III a vyšším obsahující 100Mb/s nebo 1Gb/s síťový adaptér. Měl být vybaven nejméně 256MiB operační paměti a diskem o minimální kapacitě 20GB, samozřejmostí by mělo být nasazení RAID. Základem síťové infrastruktury by měl být minimálně 100Mb/s nejlépe však 1Gb/s switch obsahující dostatečné množství portů pro připojení boot serveru a všech stanic clusteru. Další částí síťové infrastruktury je kabeláž odpovídající rychlosti použitých switchů a síťových adaptérů. Stanice clusteru by měly být osazeny procesorem Pentium III a lepším, minimálně 256MiB operační paměti a nejméně 100Mb/s síťovým adaptérem. Pevný disk není nutný, protože systém i veškerá data jsou uloženy na boot serveru.

Druh použitého hardware pro stanice má klíčový význam pro celkový výkon clusteru. Používat velmi zastaralý nevýkonný hardware přináší velmi malý nárůst výkonu ve srovnání s nákupem nového počítače.

9.2 Doporučení pro výstavbu sítě

Pro síť která bude využita pro provoz clusteru platí, že by měly být využity co nejvýkonnější switche. Dnes již zastaralé 10Mb/s huby a switche jsou použitelné maximálně pro cluster skládající se z 2 až 4 stanic při provozování programů, které neprovádí velké množství diskových operací. Při využití 100Mb/s síťových prvků je vhodné používat pouze switche a množství připojených stanic může být podle náročnosti provozované aplikace až 50. Využití 1Gbit/s síťových prvků dovoluje sestavit cluster z více než 200 stanic. Pro optimální využití síťové kapacity je rovněž nutné použít vyhovující kvalitní kabeláž.

S ohledem na připojení důležitých prvků clusteru (DHCP server a boot server) je možné, aby síť obsahovala více DHCP serverů na jednotlivých segmentech a jeden boot server. Více je tato možnost popsána v příloze PI.

ZÁVĚR

V práci je popsán výběr vhodného projektu pro provoz clusteru na platformě Linux a implementace tohoto řešení s ohledem na využití pro potřeby UTB ve Zlíně. Podle předem daných kritérií bylo přistoupeno k výběru projektu umožňujícího vytvořit SMP cluster.

Jedním z perspektivních projektů v oblasti SMP je Kerrighed, který je zaměřen na podporu mnoha nových vlastností, které umožní využít celý cluster, jakoby se jednalo o běžný víceprocesorový počítač. První verze Kerrighed (verze do 1.0) přinášely nové funkce. Naproti tomu v současné době (verze 2.0.0 a vyšší) je vývoj projektu Kerrighed zaměřen zejména na vyřešení chyb a na vytvoření stabilní verze použitelné v produkčním prostředí.

Součástí práce je kompletní návod na sestavení sítě pro cluster obsažený v příloze P I, návod na instalaci podpůrného vybavení obsažený v příloze P II a vytvoření operačního systému pro stanice clusteru, kterému je věnována příloha P III. Návod na instalaci zohledňuje aktuální podmínky, za kterých by probíhal provoz clusteru na UTB ve Zlíně. Jako optimální varianta byla zvolena možnost zavedení operačního systému pro stanice clusteru ze sítě s využitím PXE. Rovněž zvolené řešení a projekt Kerrighed poskytují možnosti efektivní centrální správy celého clusteru.

Dále jsou součástí práce manuálové stránky nástrojů Kerrighed, které byly přeloženy do českého jazyka, a jsou obsaženy v přílohách P IV až P XI.

Součástí práce je také krátká pasáž věnující se programování paralelních aplikací a aplikací využívajících funkcí Kerrighed. Na jejím základě byly vytvořeny testovací aplikace, které byly využity pro zhodnocení nárůstu výkonu při využití clusteru.

Při práci s projektem Kerrighed se vyskytl problém komplikující korektní migraci procesů mezi stanicemi, který se nepodařilo vyřešit. To vedlo k tomu, že nebylo možné plně otestovat nárůst výkonu v požadovaných aplikacích. Uspokojivé vyřešení tohoto problému se dá očekávat v budoucích verzích.

ZÁVĚR V ANGLIČTINĚ

This graduation thesis describes the suitable project selection for the cluster operation on Linux platform and implementation of this solution with respect to the use for the UTB benefit in Zlín. According to advanced given criteria was chosen this project because it enables to create SMP cluster.

One of the perspective projects in area of SMP is Kerrighed which targets to support many new features that allow using the whole cluster, as a normal multiprocessor computer. The first Kerrighed version (versions till 1.0) brought new functions. On the other hand today (version 2.0.0 and higher) Kerrighed project development is currently is aimed to solve the errors and to create stable version, that is usable in a production environment.

A part of the work is complete instructions for building network for cluster that is contained in the insertion P I, direction for installation promotive equipment, that is contained in insertion P II and for creation operating system for the cluster that is given an insertion P III. Installation instructions take into account the actual conditions for the cluster running on UTB in Zlín. As the optimal option was chosen boot of operating system for the cluster stations from the network using PXE. As well the select solution and Kerrighed project provide effective central management of the whole cluster.

Then a part of the work is also manual pages of tools of Kerrighed which were translated into Czech language and which are contained in insertions P IV to P XI.

A part of this work is also a short passage dealing with the parallel applications programming and applications using Kerrighed functions. On their basis were developed test applications, which were used for valorization increase of output with the using cluster using.

On working with Kerrighed was occurred problem, which complicates correct migration of processes between the stations, which is not managed to solve. This problem made that it was not possible to fully test the performance increase in the required applications. The satisfactory solution could be expected in next versions.

SEZNAM POUŽITÉ LITERATURY

[1] Počítačový cluster [online]. 2007 [cit. 2008-04-20]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Počítačový_cluster>.

[2] *Symmetric multiprocessing* [online]. 2009 [cit. 2009-03-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Symmetric_multiprocessing>.

[3] Non-Uniform Memory Access [online]. 2009 [cit. 2009-03-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access>.

[4] Massive parallel processing [online]. 2009 [cit. 2009-03-12]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Massively_parallel_processing>.

[5] *GPGPU* [online]. 2009 [cit. 2009-03-15]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Gpgpu>.

[6] Výpočetní cluster [online]. 2007 [cit. 2008-04-20]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Výpočetní_cluster>.

[7] The Linux Clustering Information center [online]. [2004] [cit. 2009-01-26]. Dostupný z WWW: <http://lcic.org/>.

[8] Beowulf (computing) [online]. 2009 [cit. 2009-01-26]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Beowulf_cluster>.

[9] *Beowulf.org: the Beowulf Cluster Site* [online]. c2008 [cit. 2009-01-26]. Dostupný z WWW: <http://www.beowulf.org/>.

[10] Linux [online]. 2009 [cit. 2009-02-25]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Linux>.

[11] JELÍNEK, Lukáš. Jádro systému Linux : kompletní průvodce programátora. [s.l.] : [s.n.], 2008. 686 s. ISBN 978-80-251-2084-2.

[12] LOTTIAUX, Renaud. OpenMosix, OpenSSI and Kerrighed [online]. 2006 [cit. 2009-01-26]. Dostupný z WWW: <http://hal.inria.fr/inria-00070604/en/>.

[13] OpenMosix [online]. c2008 [cit. 2009-01-26]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Openmosix>.

[14] *The openMosix Project* [online]. 2008 [cit. 2009-01-26]. Dostupný z WWW: <http://openmosix.sourceforge.net/>.

[15] OpenSSI [online]. 2008 [cit. 2009-01-27]. Dostupný z WWW: <http://en.wikipedia.org/wiki/OpenSSI>.

[16] *OpenSSI* [online]. 2006 [cit. 2009-01-27]. Dostupný z WWW: <http://www.openssi.org/>.

[17] Kerrighed [online]. 2008 [cit. 2009-01-26]. Dostupný z WWW: <http://www.kerrighed.org/wiki/index.php/Main_Page>.

[18] SPECTOR, David. Building Linux Clusters. [s.l.] : O'Reilly, 2000. 332 s. ISBN 1565926250.

[19] General Kerrighed Architecture [online]. c2007 [cit. 2009-03-14]. Dostupný z WWW: <http://kerrighed.org/wiki/index.php/KerrighedArch>.

[20] UserDoc-Kerrighed [online]. 2009 [cit. 2009-04-27]. Dostupný z WWW: <http://kerrighed.org/wiki/index.php/UserDoc>.

[21] DOBIÁŠ, Ladislav. Úvod do programování pod OS Unix [online]. 1997 [cit. 2009-04-20]. Dostupný z WWW: <http://docs.linux.cz/programming/c/c_dobias/>.

[22] *MPlayer - The Movie Player:* [online]. 2009 [cit. 2009-05-04]. Dostupný z WWW: <http://www.mplayerhq.hu>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

BIOS Basic Input-Output Systém. Je to firmware počítače typu IBM PC implementující základní vstupně–výstupní funkce.

CPU Central Processing Unit. Jde o základní součást počítače určená k provádění výpočtů.

CUDA Compute Unified Device Architecture. Jde o způsob využití grafických karet nVidia.

DHCP Dynamic Host Configuration Protocol. Jde o protokol používaný pro automatické přidělování IP adres jednotlivým osobním počítačům v počítačových sítích pro zjednodušení jejich správy.

EPM Enhanced Process Manager. Správce procesů.

FIFO First In, First Out. Je to fronta používaná ke komunikaci, kdy vložené prvky z fronty vychází ve stejném pořadí, jak byly vloženy.

FS FileSystem. Systém souborů.

HA High Performance Availability. Cílové využití clusteru.

HPC High Performance Computing. Cílové využití clusteru.

IPC InterProcess Communication. Meziprocesová komunikace.

KDDM Kerrighed Distributed Data Manager. Mechanismus distribuované správy dat.

liveCD Jde o operační systém uložený na bootovatelném CD, který z něj může být spuštěn bez nutnosti instalace.

MM Memory Management. Správce paměti.

MPI Message Passing Interface. Komunikační rozhraní užívané u clusterů.

MPP Masively Parallel Processing. Architektura clusterů.

NFS Network File Systém. Je to síťový protokol určený pro vzdálený přístup k souborům přes počítačovou síť.

NTPL Native Posix Threat Library. Knihovna splňující specifikaci posix pro vytváření přenositelných vícevláknových aplikací.

NUMA Non-Uniform Memory Access. Architektura víceprocesorových počítačů.

PID Process IDentifier. Celočíselný identifikátor procesu.

POSIX Portable Operating System Interface for uniX. Přenositelné rozhraní pro operační systémy.

POST Power-On Self-Test. Jde o společný název pro sekvenci prováděnou před startem operačního systému.

PVM Paralel Virtual Machine. Softwarový nástroj pro provoz clusterů.

PXE Preboot eXecution Environment. Je to prostředí, které je využíváno k zavedení operačního systému ze sítě nezávisle na úložných zařízeních instalovaných v počítači.

RAID Redundant Array of Inexpensive Disks. Tzv. vícenásobné pole levných disků, které slouží ke zvýšení výkonu a zabezpečení uložených dat.

RPC Remote Procedure Calling. Protokol umožňující volat části programů na vzdáleném stroji.

SMP Symmetric Multi-Processing. Architektura víceprocesorových počítačů.

SSI Single Systém Image. Architektura clusterů.

SUID Set User IDentifier. Příznak umožňující spustit program s efektivními právy vlastníka souboru.

TFTP Trivial File Transfer Protocol. Jde o jednoduchý protokol pro přenos souborů, který obsahuje jen základní funkce protokolu FTP.

TID Thread IDentifier. Celočíselný identifikátor vlákna.

SEZNAM OBRÁZKŮ

Obr. 1: Blokové schéma typického SMP systému 12
Obr. 2: Blokové schéma systému NUMA 12
Obr. 3: Rozdíl mezi tradičním SMP systémem, MPP clusterem a Kerrighed clusterem 24
Obr. 4: Blokové schéma architektury Kerrgihed 25
Obr. 5: Snímek obrazovky htop zobrazující test bez povolené migrace procesů 33
Obr. 6: Snímek obrazovky htop zobrazující test s povolenou migrací procesů 33
Obr. 7: Rodičovský proces čekající na potomka 38
Obr. 8: Funkce mutexu 43
Obr. 9: Topologie základní sítě clusteru 59
Obr. 10: Topologie sítě clusteru se stávajícím DHCP serverem 59

SEZNAM PŘÍLOH

P I Návod na sestavení sítě pro cluster

P II Instalace a nastavení boot serveru

P III Vytvoření obrazu systému pro stanice

P IV Manuálová stránka kerrighed_nodes(5)

P V Manuálová stránka krgadm(1)

P VI Manuálová stránka kerrighed_capabilities (7)

P VII Manuálová stránka krgcapset(1)

P VIII Manuálová stránka krgcapset(2)

P IX Manuálová stránka migrate(1)

P X Manuálová stránka migrate(2)

P XI Manuálová stránka migrate_self(2)

P XII CD obsahující elektronickou verzi diplomové práce a následující adresáře s obsahem:

zdroje – iso obraz liveCD kerrighed, iso obrazinstalačního CD Debian GNU/Linux a zdrojové kódy linux-2.6.20

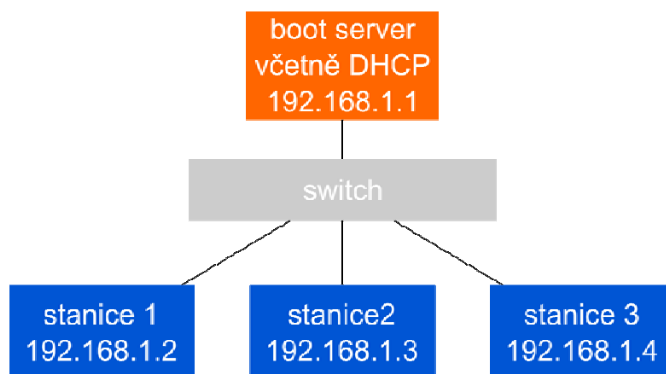
zdojove_kody – zdrojove kody ukázkových a testovacích aplikací.

konfigurace – konfigurační soubory zmiňované v práci.

PŘÍLOHA P I: NÁVOD NA SESTAVENÍ SÍTĚ PRO CLUSTER

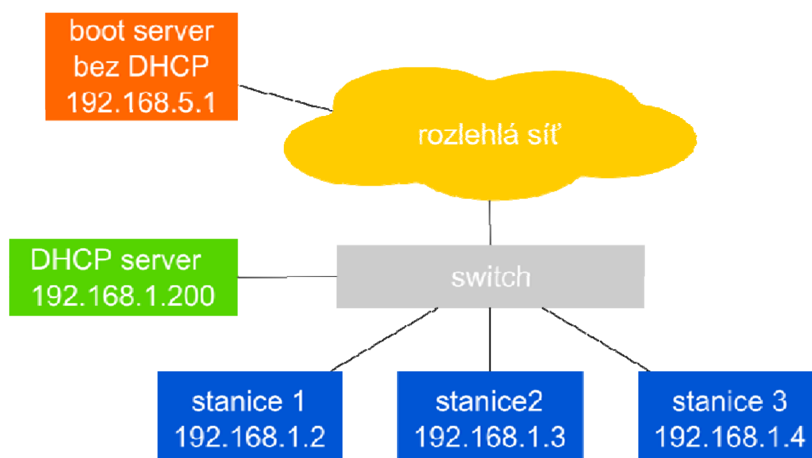
Před samotnou instalací zvolte vhodný rozsah IP adres použitých pro cluster. V návodu bude aplikována síťová adresa 192.168.1.0/24, které odpovídá rozsah využitelných adres od 192.168.1.1 až 192.168.1.254, síťová maska 255.255.255.0 a všesměrová adresa 192.168.1.255. Pro boot server bude přidělena adresa 192.168.1.1, pro router a DNS server 192.168.1.200 a stanicím budou přidělovány adresy z rozsahu 192.168.1.2 až 192.168.1.199.

Pro jednoduchost není zakresleno připojení k internetu a další infrastruktuře. Server a stanice propojte pomocí tzv. přímých kabelů do switche. V případě, že součástí boot serveru bude i DHCP, musí být topologie sítě hvězdicová, podle obrázku (*Obr. 9*).



Obr. 9: Topologie základní sítě clusteru

V případě využití jiného zařízení (například routeru) jako DHCP serveru může být topologie sítě složitější a samotný boot server nemusí být na stejném segmentu sítě jako stanice (bude oddělen rozlehlou sítí), podle obrázku (Obr. 10).



Obr. 10: Topologie sítě clusteru se stávajícím DHCP serverem

PŘÍLOHA P II: INSTALACE A NASTAVENÍ BOOT SERVERU

Pro instalaci Debian GNU/Linux na boot server je nejlepší využít tzv. netinstall instalační médium. Jde o instalační médium, které obsahuje pouze instalátor a veškeré instalované balíčky jsou stahovány v aktuální verzi z online repositářů. Obraz netinstall instalačního média je možné získat na adrese http://debian.org/distrib/netinst. Po vypálení instalačního obrazu na médium začněte jeho vložením do mechaniky boot serveru a restartuje jej (je nutné mít v BIOS zapnutou podporu zavádění z CD/DVD). Po skončení POST následuje spuštění instalátoru.

Během instalace oproti výchozím hodnotám zvolte jako součásti systému pouze **základní systém** a **DNS server**. Tím je instalace základního systému pro boot server hotova. Následuje instalace potřebných aplikací a nastavení serveru.

Po instalaci se přihlaste jako uživatel **root**, pod nímž budou prováděny všechny následující kroky. Editujte soubor `/etc/apt/sources.list` (například pomocí editoru pico) a zakomentujte (vepsáním znaku # na začátek řádku) řádek začínající `deb cdrom` a provedené změny uložte. Pomocí příkazu `apt-get update` aktualizujte seznam balíčků dostupných v online repositářích.

Nainstalujte potřebné balíčky obsahující vývojové nástroje a servery použitých služeb pomocí příkazu:

```
apt-get install mc tftp-hpa nfs-kernel-server ssh bzip2 debootstrap
syslinux psmisc automake make autoconf libtool rsync gcc-3.4 pkg-config
lsb-release xmlto
```

Během instalace balíčků odpovězte na všechny otázky kladně.

Při konfiguraci služeb serveru zohledněte zvolený rozsah IP adres a MAC adresy síťových adaptérů jednotlivých stanic clusteru, s ohledem na topologii sítě.

Pro zprovoznění služby TFTP vytvořte konfigurační soubor `/etc/default/atftpd`, který bude obsahovat následující:

```
USE_INETD=false

OPTIONS="--daemon --port 69 --tftpd-timeout 300 --retry-timeout 5 --
mcast-port 1758 --mcast-addr 239.239.239.0-255 --mcast-ttl 1 --maxthread
100 --verbose=5 /srv/tftp"
```

Pro dokončení nastavení TFTP serveru zakomentujte v souboru `/etc/inetd.conf` řádek, který začíná `tftp` a vytvořte kořenový adresář služby pomocí příkazu:

```
mkdir /srv/tftp
```

Pokračujte nastavením NFS serveru, které provedete vytvořením souboru `/etc/exports`, který bude obsahovat následující:

```
/NFSROOT/kerrighed
192.168.1.0/255.255.255.0(ro,async,no_root_squash,no_subtree_check)
/NFSROOT/kerrighed/tmp
192.168.1.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)
/NFSROOT/kerrighed/var
192.168.1.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)
/NFSROOT/kerrighed/root
192.168.1.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)
/NFSROOT/kerrighed/etc
192.168.1.0/255.255.255.0(rw,sync,no_root_squash,no_subtree_check)
```

Dále pomocí příkazu `mkdir /NFSROOT` vytvořte adresář, ve kterém bude uložen sdílený svazek systému provozovaného na stanicích clusteru.

Nastavení boot serveru dokončete editací nastavení síťového rozhraní v souboru `/etc/network/interfaces`, který poté bude obsahovat:

```
auto lo eth0
iface lo inet loopback
allow-hotplug eth0
iface eth0 inet static
        address 192.168.1.1
        netmask 255.255.255.0
```

Nyní jsou nastavení všech služeb dokončena a zbývá služby restartovat pomocí příkazů:

```
killall inetd && /usr/sbin/inetd
/etc/init.d/atftpd restart
/etc/init.d/nfs-kernel-server restart
killall portmap && portmap
```

V případě, že bude DHCP server součástí boot serveru jej nainstalujte příkazem:

```
apt-get install mc dhcp3-server
```

Dále nastavte DHCP server vytvořením souboru `/etc/dhcp3/dhcpd.conf`, který bude obsahovat:

```
option dhcp-max-message-size 2048;
use-host-decl-names on;
deny unknown-clients;
deny bootp;
option domain-name "cluster";
option domain-name-servers 192.168.1.200;       #adresa DNS serveru
subnet 192.168.1.0 netmask 255.255.255.0 {
```

```
  option routers 192.168.1.200;                #adresa výchozího routeru
  option broadcast-address 192.168.1.255
}

group {
  filename "pxelinux.0";      #cesta relativni ke korenu TFTP serveru
  option root-path "192.168.1.1:/NFSROOT/kerrighed"; #parametr pro jadro
  #deklarace stanic clusteru
  host node1   { fixed-address 192.168.1.2;    #deklarace prvni stanice
               hardware ethernet 01:02:03:04:05:06; }
  host node2   { fixed-address 192.168.1.3;    #deklarace druhe stanice
               hardware ethernet 01:02:03:04:05:07; }
  next-server 192.168.1.1;    #adresa TFTP serveru
}
```

Pro aplikaci nového nastavení restartujte DHCP server příkazem:

```
/etc/init.d/dhcp3-server restart
```

PŘÍLOHA P III: VYTVOŘENÍ OBRAZU SYSTÉMU PRO STANICE

Postup vytvoření systému provozovaného na stanicích clusteru se skládá z vytvoření základního systému, kompilace jádra Linux s podporou Kerrighed a kompilace utilit pro ovládání a správu clusteru.

Systém provozovaný na stanicích clusteru bude založen opět na Debian GNU/Linux konkrétně ve verzi Sid.

Použít můžete následující postup:

Vytvoření základní adresářové struktury obsahující základní použité aplikace a knihovny:

```
debootstrap sid /NFSROOT/kerrighed http://ftp.debian.org/debian
```

Přepnutí kořenového adresáře v aktuálním terminálu do struktury budoucího systému:

```
chroot /NFSROOT/kerrighed
```

Pomocí příkazu `passwd` nastavte nové heslo uživatele root pro stanice clusteru.

Připojte systém souborů `/proc`:

```
mount -t proc none /proc
```

Aktualizace seznamu balíčků:

```
apt-get update
```

Instalace dalších důležitých aplikací, knihoven a vývojových nástrojů:

```
apt-get install dhcp3-common nfs-common nfsbooted psmisc automake make
autoconf libtool libncurses5-dev rsync gcc-3.4 pkg-config lsb-release
xmlto openssh htop
```

Vytvoření symbolického odkazu pro gcc kompilátor na instalovaný gcc verze 3.4:

```
ln -s /usr/bin/gcc-3.4 /usr/bin/gcc
```

Souboru `/etc/fstab` upravte tak, aby obsahoval následující:

```
none         /proc  proc   defaults      0 0
none         /sys sysfs    defaults      0 0
#
# NFSROOT
192.168.0.1:/NFSROOT/kerrighed/etc  /dev  nfs rw,hard,nolock 0 0
192.168.0.1:/NFSROOT/kerrighed/var  /var  nfs rw,hard,nolock 0 0
192.168.0.1:/NFSROOT/kerrighed/tmp  /tmp  nfs rw,hard,nolock 0 0
192.168.0.1:/NFSROOT/kerrighed/root /root nfs rw,hard,nolock 0 0
#
#TMPFS
none                                /var/run tmpfs defaults     0 0
```

Vytvoření symbolického odkazu na skript, zajišťující připojení všech svazků typu NFS deklarovaných v souboru `/etc/fstab`:

```
ln -sf /etc/network/if-up.d/mountnfs /etc/rcS.d/S42mountnfs
```

Přepnutí do adresáře se zdrojovými kódy:

```
cd /usr/src
```

Stažení zdrojových kódů jádra Linux:

```
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.tar.bz2
```

Stažení zdrojových kódů Kerrighed:

```
wget http://kerrighed.gforge.inria.fr/kerrighed-latest.tar.gz
```

Rozbalení zdrojových kódů jádra Linux:

```
tar -jxf linux-2.6.20.tar.bz2
```

Rozbalení zdrojových kódů Kerrighed:

```
tar -zxf kerrighed-latest.tar.gz
```

Přepnutí do adresáře se zdrojovými kódy Kerrighed:

```
cd kerrighed-*
```

Spuštění konfiguračního skriptu pro překlad Kerrighed:

```
./configure –with-kernel=/usr/src/linux-2.6.20
```

Aplikace úprav jádra Linux:

```
make patch
```

Vytvoření výchozí konfigurace:

```
make defconfig
```

Výchozí konfigurace nemusí vždy vyhovovat použitému hardware. Je nutné zkontrolovat, přítomnost podpory zejména pro použitý síťový adaptér. Konfigurace je k nalezení v souboru `/usr/src/linux-2.6.20/.config`, případně je možné využít příkazu `make menuconfig` spuštěného v adresáři `/usr/src/linux-2.6.20`.

Překlad samotného jádra Linux:

```
make kernel
```

Překlad Kerrighed, respektive komponent pro jeho správu a ovládání:

```
make
```

Instalace nově přeloženého jádra:

```
make kernel-install
```

Instalace přeložených nástrojů Kerrighed:

```
make install
```

Opusťte chroot prostředí pomocí příkazu `exit`.

Zkopírování obrazu přeloženého jádra Linux do adresáře TFTP serveru:

```
cp /NFSROOT/kerrighed/boot/vmlinuz-* /srv/tftp
```

Zkopírování zavaděče pxelinux do adresáře TFTP serveru:

```
cp /usr/lib/syslinux/pxelinux.0 /srv/tftp
```

Vytvoření adresáře obsahujícího konfigurační soubory pro pxelinux:

```
mkdir /srv/tftp/pxelinux.cfg
```

V adresáři `/srv/tftp/pxelinux.cfg` vytvořte soubor `default` obsahující:

```
default cluster
label cluster
        kernel /vmlinuz-2.6.20-krg
        append console=tty1 root=/dev/nfs
nfsroot=192.168.1.1:/NFSROOT/kerrighed ro ip=dhcp pci=nommconf
```

Boot server a systém pro stanice jsou nyní kompletní a připraveny k provozu.

PŘÍLOHA P IV: MANUÁLOVÁ STRÁNKA KERRIGHED_NODES(5)

Název

kerrighed_nodes — Soubor obsahující seznam stanic a čísla sezení Kerrighed clusteru.

Popis

Soubor kerrighed_nodes obsahuje seznam uzlů náležících ke Kerrighed clusteru. Každý název uzlu je na samostatném řádku a je následován logickým číslem stanice a jménem síťového zařízení, které bude využito Kerrighed ke komunikaci.

Položka session obsahuje identifikátor clusteru. Pokud je několik kerrighed clusterů připojeno ke stejnému síťovému switchi, může být definováno více identifikátorů sezení k zamezení kolizí. Identifikátor sezení je v rozsahu integer.

Položka nbmin slouží k definování počtu stanic, při jehož dosažení při startu je Kerrighed cluster automaticky inicializován. Tato položka je volitelná. Pokud je tato položka nastavena, není k inicializaci clusteru nutné využívat **krgadm**. Během fáze startu systému na stanicích clusteru modul kerrighed čeká, než bude stanovený počet stanic připraven. Po dosažení požadovaného počtu stanic je Kerrighed automaticky inicializován.

Příklad 1. kerrighed_nodes

```
session=1
nbmin=4
node1:0:eth0
node2:1:eth0
node3:2:eth0
node4:3:eth0
```

Výchozí umístění pro soubor kerrighed_nodes je v adresáři /etc/kerrighed.

Soubory

/etc/kerrighed_nodes

Související

kerrighed_session (5), krgadm (1)

PŘÍLOHA P V: MANUÁLOVÁ STRÁNKA KRGADM(1)

Název

`krgadm` — Start, zastavení, připojení nebo opuštění clusteru

Přehled

```
krgadm [ -h | --help ]
```

```
krgadm cluster { status | start | poweroff | reboot } [ -s | --
subsessions ] [ -n | --nodes ]
```

```
krgadm nodes { status | add | del | poweroff | reboot | fail | swap | ban
| unban } [ -s | --subsessions ] [ -n | --nodes ]
```

Popis

Pogram krgadm dovoluje uživateli pomocí jediného příkazu provádět správu clusteru, od startu po vypnutí, přes přidávání a odebírání stanic až po monitorování stavu clusteru.

Při každém spuštění krgadm musí následovat příkaz a parametry, které přísluší k příkazu.

`cluster`

> Start, zastavení a sledování stavu clusteru.

`nodes`

> Správa stanic clusteru.

Parametry

`-h, --help`

> Výpis nápovědy a ukončení

Parametry pro režim správy clusteru a stanic

`-s, --subsession`

> Identifikátor sezení.

`-n, --nodes`

> Seznam uzlů. Jednotlivé hodnoty jsou odděleny čárkou.

PŘÍLOHA P VI: MANUÁLOVÁ STRÁNKA KERRIGHED_CAPABILITIES (7)

Název

`kerrighed_capabilities` — Přehled schopností Kerrighed.

Popis

Kerrighed poskytuje sady schopností, které umožňují administrátorovi a uživatelům definovat schopnosti svých procesů v rámci poskytovaných mechanismů. Mechanismy poskytované Kerrighed, dostupné v rámci celého clusteru, jsou rozděleny do několika samostatných jednotek, které mohou být nezávisle na sobě zapínány a vypínány.

Seznam schopností

Následující schopnosti jsou v současné době plně implementovány:

CAP_CHANGE_KERRIGHED_CAP

Povoluje změnu schopností.

CAP_CAN_MIGRATE

Povoluje migraci procesů.

CAP_DISTANT_FORK

Tato schopnost je využívána systémovým voláním fork pro rozhodnutí, jestli má být nový proces spuštěn na vzdálené stanici. Úspěch této operace ale není zaručen.

CAP_SEE_LOCAL_PROC_STAT

Dovoluje pohled na soubory v `/proc` příslušející lokální stanici na rozdíl od globálního `/proc` pro celý cluster.

Schopnosti procesu

Každý proces má přiřazeny čtyři sady schopností, které mohou obsahovat buď žádnou nebo libovolnou kombinaci schopností:

Effective

Efektivní schopnosti aplikované jádrem operačního systému na procesy.

Pertmitted

Povolené schopnosti, které může proces převzít (omezuje nadmnožinu efektivních, dědičných a dědičných povolených sad). Pokud proces některé schopnosti ztratí, nemůže je už znovu nabýt

Inheritable effective

Efektivní zděděné schopnosti stejného typu jako efektivní, aplikované na nový proces při volání `execve()`, které má na starost spuštění programu pomocí ukazatele na jeho spustitelný soubor.

Inherirable permitted

Povolené zděděné schopnosti stejného typu jako povolené, aplikované na nový proces při volání funkce `execve()`.

Přenos schopností

Během vykonávání volání fork jádro vypočítá novou schopnost procesu za pomoci následujícího algoritmu:

P'(povolená) = (P(dědičná povolená) & F(povolená)) | F(povolená)P'(efektivní) = P(dědičná efektivní) & F(efektivní) & P'(povolená)P'(dědičná povolená) = P(dědičná povolená)

Kde :

P Značí hodnotu schopnosti procesu nastavenou před spuštěním

P' Značí hodnotu schopnosti nastavenou po spuštění

F Značí práva souboru

Soubory

`/etc/kerrighed_nodes`

Tento soubor obsahuje seznam stanic použitých v kerrighed clusteru. Více informací je dostupných v manuálové stránce pro kerrighed_nodes(5).

Související

krgcapset(1)

PŘÍLOHA P VII: MANUÁLOVÁ STRÁNKA KRGCAPSET(1)

Název

`krgcapset` — Nastavení a modifikace schopností procesu.

Přehled

```
krgcapset [ -h | --help ]
krgcapset [ -s | --show ]
krgcapset [ -f | --force ] [ -k | --pid ] { SET { [ + | - ] CAPABILITY
LIST | OCTAL VALUE }...}
```

Sady

`-e, --effective`

Nastavuje efektivní schopnosti.

`-p, --permitted`

Nastavuje povolené schopnosti.

`-d, --inheritable-effective`

Nastavuje dědičné efektivní schopnosti.

`-i, --inheritable-permitted`

Nastavuje dědičné povolené schopnosti.

Popis

Kerrighed poskytuje systém schopností, které umožňují administrátorovi a uživatelům definovat způsob chování procesů. Vlastnosti dostupné v celém Kerrighed clusteru jsou rozděleny do několika rozdílných jednotek, které mohou být nezávisle zapínány nebo vypínány. Každý proces má čtyři sady schopností:

- Efektivní
- Povolené
- Dědičné efektivní
- Dědičné povolené

Více detailů je dostupných v manuálové stránce kerrighed_capabilities(7). Krgcapset modifikuje schopnosti volaného procesu.

Parametry

`-h, --help`

Vypíše nápovědu a skončí.

`-s, --show`

Vypíše schopnosti pro volaný proces.

`-f, --force`

Odebrání schopnosti CHANGE_KERRIGHED_CAP bez potvrzení uživatele.

Příklady

`krgcapset --show`

Vypíše schopnosti volaného procesu.

`krgcapset --effective +CAN_MIGRATE`

Přidá schopnost CAN_MIGRATE do sady efektivních schopností volaného procesu.

`krgcapset --effective -CAN_MIGRATE`

Odebírá schopnost CAN_MIGRATE ze sady efektivních schopností volaného procesu.

`krgcapset --effective 07`

Přidá mezi efektivní schopnosti CHANGE_KERRIGHED_CAP, USE_CONTAINERS, CAN_MIGRATE.

Související

kerrighed_capabilities(7)

PŘÍLOHA P VIII: MANUÁLOVÁ STRÁNKA KRGCAPSET(2)

Název

`krgcapset` — Nastavení a modifikace schopností procesu.

Přehled

```
#include <libkerrighed.h>
int krg_capset(   *new_caps);
struct krg_cap_t        *new_caps;
```

Popis

Systémové volání krg_capset modifikuje schopnosti volajícího procesu pomocí zaslání schopnosti skrze new_caps. Struktura krg_cap_t obsahuje následující pole:

```
struct krg_cap_t {
  int   krg_cap_effective;
  int   krg_cap_permitted;
  int   krg_cap_inheritable_permitted;
  int   krg_cap_inheritable_effective ;
};
```

Více detailů je dostupných v manuálové stránce krg_capabilities(7).

Návratová hodnota

Při úspěchu je vrácena 0. Při chybě je vrácena -1, a je vhodně nastavena chybová proměnná.

Chyby

EPERM

Proces nemá právo pro změnu vlastní schopnosti.

Související

krgcapget (1), kerrighed_capabilities(7)

PŘÍLOHA P IX: MANUÁLOVÁ STRÁNKA MIGRATE(1)

Název

`migrate` — Migruje proces na zadanou stanici.

Přehled

`migrate pid nodeid`

Description

Program migrate migruje proces se zadaným PID na stanici clusteru se zadaným nodeid.

Soubory

`/etc/kerrighed_nodes`

Soubor obsahuje seznam stanic v Kerrighed clusteru. Více detailů je dostupných v manuálové stránce kerrighed_nodes(5).

Související

krg_capset(1)

PŘÍLOHA P X: MANUÁLOVÁ STRÁNKA MIGRATE(2)

Název

`migrate` — Migruje proces na zadanou stanici.

Přehled

```
#include <libkerrighed.h>
int migrate(pid,node_id);
pid_t pid;
int   node_id;
```

Popis

Volání `migrate` migruje proces se zadaným PID na stanici clusteru se zadaným nodeid.

Návratová hodnota

Při úspěchu je vrácena 0. Při chybě je vrácena -1 a je vhodně nastavena chybová proměnná.

Chyby

ESRCH

> Proces nebo skupina procesů se zadaným PID neexistuje

EPERM

> Proces nemá povolení k migraci zadaného procesu. To může nastat, pokud proces nemá nastaveny správné schopnosti (více detailů je v manuálové stránce krg_capability(7)) nebo pokud byl otevřen lokální proud.

`/etc/kerrighed_nodes`

> Soubor obsahuje seznam stanic v Kerrighed clusteru. Více detailů je dostupných v manuálové stránce kerrighed_nodes(5).

Související

migrate_self(2), krg_capability(7)

PŘÍLOHA P XI: MANUÁLOVÁ STRÁNKA MIGRATE_SELF(2)

Název

`migrate_self` — Migruje volající proces na zadanou stanici.

Přehled

```
#include <libkerrighed.h>
int migrate_self(node_id);
int   node_id;
```

Popis

Volání migrate_self migruje proces, který toto volání použil na stanici clusteru se zadaným nodeid.

Návratová hodnota

Při úspěchu je vrácena 0. Při chybě je vrácena -1 a je vhodně nastavena chybová proměnná.

Chyby

EPERM

> Proces nemá právo migrovat na zvolenou stanici. To může nastat pokud nemá proces nastaveny patřičné schopnosti (více detailů je v manuálové stránce krg_capability(7)).

Soubory

`/etc/kerrighed_nodes`

> Soubor obsahuje seznam stanic v Kerrighed clusteru. Více detailů je dostupných v manuálové stránce kerrighed_nodes(5).

Soubory

`/etc/kerrighed_nodes`

> Soubor obsahuje seznam stanic v Kerrighed clusteru. Více detailů je dostupných v manuálové stránce kerrighed_nodes(5).

Související

migrate(2), krg_capability(7)