System Vproc projektu OpenSSI

Projekt z przedmiotu RSO

Michał Władysław Garcarz
mgarcarz@elka.pw.edu.pl

14 czerwca 2005

Spis treści

1 Vproc - wprowadzenie **1**
1.1 Wstęp 1
1.2 Możliwości 2

2 Architektura Vproc **2**
2.1 Wstęp 2
2.2 Dokładniejszy opis struktur 3
2.2.1 vproc 3
2.2.2 pvproc 3
2.2.3 oryginalna task_struct 4
2.2.4 rozszerzenia task_struct 5
2.3 Porównanie z tradycyjnym kernelem 6
2.4 Numery procesów 6
2.5 Śledzenie procesów 6
2.6 Przechowywanie struktur vproc i pvproc 7
2.7 Rozpraszanie procesów 7
2.8 Migracja procesów 8
2.9 Migracja procesów - dynamika 9
2.10 API dla użytkownika 10
2.10.1 Wywołania systemowe 10
2.10.2 Proc 10
2.10.3 Funkcje biblioteczne 11
2.10.4 Przykłady wykorzystania API 12
2.11 Testy wydajnościowe 12

1 Vproc - wprowadzenie

1.1 Wstęp

W celu umożliwienia stworzenia systemu rozproszonego, który pracując na wielu maszynach umożliwiałby przezroczystą dla użytkownika migrację procesów pomiędzy tymi maszynami należało stworzyć w kernelu odpowiednie struktury i mechanizmy zarządzania procesami. Mechanizmy te zostały nazwane przez twórców OpenSSI

Vproc. Rozszerzają one możliwości tradycyjnego kernela jednocześnie zapewniając jego zgodność z tymi strukturami, które nie są świadome pracy w środowisku rozproszonym.

Dokument ten skupia się na opisaniu architektury Vproc oraz różnic w zarządzaniu procesami w stosunku do standardowych kerneli linuxa. W dalszej części przedstawione zostaną praktyczne przykłady obrazujące działanie Vproc.

1.2 Możliwości

Architektura Vproc umożliwia rozproszenie (rozdzielenie):

- Procesu rodzica i dziecka
- Grupy procesów
- Procesów należących do jednej sesji
- Procesu debugującego i debugowanego
- Procesu wysyłającego i odbierającego sygnał

Dzięki temu w systemie OpenSSI umożliwiono:

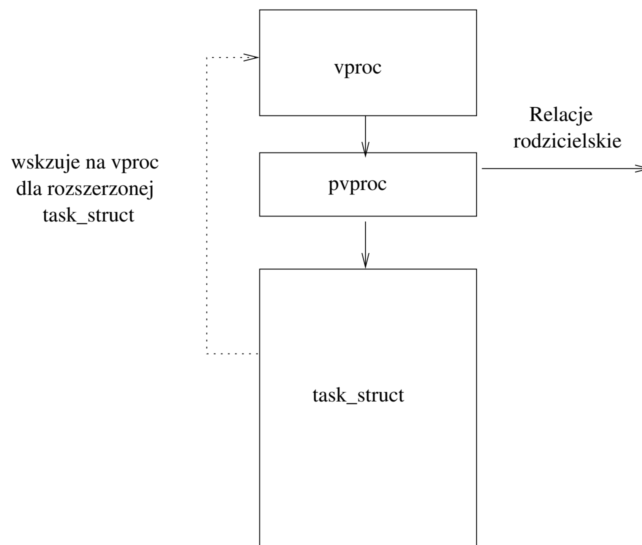
- Jeden system /proc ze wszystkimi procesami dostępny w każdym węźle klastra
- Przezroczystą migrację procesów
- Ręczna migracja procesów (pełna kontrola nad procesem migracji)
- Równoważenie obciążenia (ręczne i automatyczne)
- Przezroczysty proces debugowania
- Gwarancja poprawnej obsługi procesów w przypadku odłączania się i powtórnego dołączania do klastra jednego z węzłów.

Z faktu dostępności wspólnego systemu /proc wynika znacznie więcej. Należy dodać, że twórcy kernela stworzyli w systemie proc szereg wpisów (poprzez kernelową funkcję create_proc_entry()), które umożliwiają odczyt aktualnych parametrów związanych z całym klastrem jak i z poszczególnymi jego węzłami oraz ich modyfikacje.

2 Architektura Vproc

2.1 Wstęp

Vproc musiał zostać tak zaimplementowany aby nie kolidował z istniejącą architekturą zarządzania procesami i aby nie trzeba było w tym celu przepisywać kodu całego kernela. Dlatego jego twórcy zdecydowali się dodać pewną *wirtualną warstwę obsługi procesów*, która wskazuje na tradycyjne struktury związane z procesami (listy struktur *task_struct*). Ta warstwa składa się z list struktur typu *vproc* i *pvproc*. Dzięki temu typowe typowe wywołania systemowe czy funkcje kernela, które nie są związane z klastrem odwołują się bezpośrednio do tradycyjnych struktur procesów, natomiast te, które są przeznaczone do pracy w klastrze odwołują się do tej *wirtualnej warstwy*, która z kolei odwołuje się do tradycyjnych struktur procesów.



Rysunek 1: Architektura Vproc

Struktura *vproc* zawiera w zasadzie tylko pid procesu, co wskazuje na odpowiednią strukturę *pvproc*. W strukturze *pvproc* natomiast zawarte są wszelkie informacje związane z pokrewieństwem oraz dane związane z klastrem (np: na jakim węźle wykonuje się dany proces). Natomiast struktura *task_struct* została nieco rozbudowana, jednak wszystkie pola, które występowały w jej oryginalnym odpowiedniku występują i tutaj (dzięki temu zachowano kompatybilność). Można zauważyć, że niektóre pola ze struktury *pvproc* (związane z pokrewieństwem) dublują podobne pola ze struktury *task_struct*. Przy obsłudze systemu klastrowego takie wpisy z *pvproc* mają pierwszeństwo nad tymi z *task_struct*.

2.2 Dokładniejszy opis struktur

2.2.1 vproc

Ważniejsze pola tej struktury to:

- *vp_pid* - pid procesu
- *vp_ref_cnt* - licznik referencji
- *vp_pid_hash* - struktura dla szybkiego wyszukiwania (hash)
- *vp_next* - wskaźnik na następny obiekt *vproc* (tylko dla VPROC_DEBUG)

Normalnie pole *vp_next* nie jest potrzebne, gdyż zazwyczaj wyszukiwanie i przeglądanie listy procesów odbywa się przy użyciu list związanych ze strukturami *task_struct*, z których są odpowiednie wskaźniki na strukturę *pvproc*.

2.2.2 pvproc

Ważniejsze pola tej struktury to:

- *pvp_pproc* - wskaźnik na rzeczywistą strukturę opisującą proces (*task_struct*)
- *pvp_head_child1* - wskaźnik na pierwsze dziecko rodzica
- *pvp_head_pgrpl* - wskaźnik na lidera grupy procesów
- *pvp_head_session1* - wskaźnik na lidera sesji
- *pvp_head_head_octlist* - wskaźnik na listę dzieci
- *pvp_ppid* - pid aktualnego rodzica
- *pvp_oppid* - pid oryginalnego rodzica
- *pvp_sid* - pid lidera sesji
- *pvp_pgid* - pid lidera grupy
- *pvp_ppid* - pid rodzica
- *pvp_fromnode* - węzeł na którym wykonuje się proces
- *pvp_tonode* - węzeł na który proces się przenosi
- *pvp_flag* - atrybuty procesu (np: proces kontroluje terminal, jest liderem grupy/sesji, kończy się, jest origin node, duchem, jest procesem systemowym)
- *pvp_wstate* - stan procesu (tyle ile potrzeba wiedzieć, gdy proces wykonuje się gdzie indziej i nie mamy podobnego pola z *task_struct*, ponieważ wtedy nie posiadamy *task_struct*)

Do tego należało by doliczyć jeszcze mniej istotne pola związane z pokrewieństwem, zamkami, statystykami czy obsługą sygnałów.

Dla procesów, które fizycznie nie są wykonywane na naszym węźle *pvp_pproc* jest równe NULL. Istnienie pola *pvp_tonode* jest konieczna w celu wyeliminowania niebezpieczeństwa niespójności danych przy migracji procesu.

Należy również pamiętać o szeregu list łączących powyższe struktury (chociaż znacznie uproszczonych w stosunku do tych, które są zaimplementowane dla *task_struct*) w celu szybkiego wyszukiwania procesów.

2.2.3 oryginalna task_struct

Jest to jedna z bardziej rozbudowanych struktur jądra, Oryginalna struktura *task_struct* zawiera:

- Status
- Flagi
- Opcje związane z szeregowaniem
- Na którym procesorze działa proces, na którym może działać.
- Liczniki referencji
- Priorytety

- Wskaźniki na poprzedni, następny proces
- Listy procesów (również hashujące), kolejki oczekiwania
- Numery procesów pokrewnych
- Dane opisujące terminal związany z procesem
- Dane ładowane do segmentu TSS dla każdego procesu (kontekst sprzętowy jak rejestry czy mapa bitowa uprawnień I/O). Od kernela 2.4 struktura znana jako thread (już tylko jeden segment TSS dla każdego procesora a nie dla każdego procesu).
- Informacje o systemie plików (aktualny katalog)
- Wskaźniki do deskryptorów plików
- Wskaźniki do deskryptorów obszarów pamięci
- Informacje związane z IPC (np: licznik dla SEMUNDO)
- Otrzymane sygnały, maski sygnałów, wskaźniki do procedur obsługi
- Limity (RLIMIT*)
- Szereg blokad w celu synchronizacji dostępu (głównie spinlock)

Należy przypomnieć, że status procesu może być jednym z:

- TASK_RUNNING - proces się wykonuje lub czeka na wykonanie
- TASK_INTERRUPTIBLE - proces czeka (np: na przerwanie lub zwolnienie zasobów systemowych)
- TASK_UNINTERRUPTIBLE - podobnie jak poprzedni, tylko że nie może być zbudzony przez dostarczony sygnał (do inicjacji urządzeń)
- TASK_STOPPED - proces zatrzymany (np: przez sygnał)
- TASK_ZOMBIE - proces się zakończył, ale rodzic nie zebrał statusu

Można zauważyć, że pokrywają się one częściowo z atrybutami procesu z *pvproc*.

2.2.4 rozszerzenia task_struct

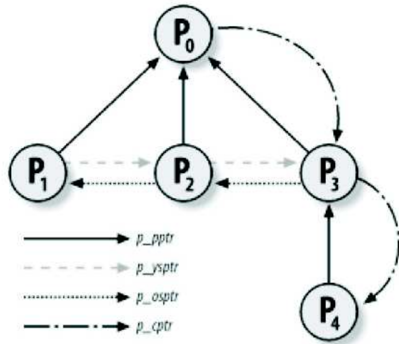
Twórcy projektu openssi rozszerzyli *task_struct* o następujące pola (najważniejsze z nich):

- *p_vproc* - wskaźnik na *vproc* danego procesu
- *p_nodetime* - czas spędzony w przestrzeni użytkownika i kernela na bieżącym węźle

Dodatkowo należy doliczyć jeszcze struktury wykorzystywane przy kopiowaniu danych poprzez sieć na poziomie kernela (np: podczas migracji).

2.3 Porównanie z tradycyjnym kernelem

Jak widać wprowadzenie struktur vproc i *pvproc* rozszerza nam możliwości systemu zachowując możliwość pracy dla procesów użytkownika (i mechanizmów kernela) nie rozpoznających środowiska rozproszonego. Dla wątków migrujących w obrębie klastra zachowane zostają wszystkie relacje rodzicielskie tak samo jak w standardowych systemach:



Rysunek 2: Relacje rodzicielskie

Możliwe relacje rodzicielskie:

- *p_opptr* - oryginalny rodzic
- *p_pptr* - rodzic
- *p_cptr* - najmłodsze dziecko
- *p_ysptr* - młodsze rodzeństwo
- *p_osptr* - starsze rodzeństwo

2.4 Numery procesów

Węzeł na którym został stworzony proces nazywany jest *origin node* i odpowiada on za śledzenie procesu (oraz jego migrację). System gwarantuje unikalność dla numerów pid procesów. Osiągnięto to przez przydzielenie każdemu węzłowi pewnej przestrzeni dostępnych numerów pid (niestety aktualnie tylko 65535 procesów). W ten sposób węzeł pierwszy tworzy procesy o numerach z przedziału (65536,2*65536). Drugi węzeł pokrywa przedział (2*65536,3*65536), n-ty (n*65536,(n+1)*65536). Dzięki takiemu podziałowi zawsze wiadomo, który węzeł odpowiada za jaki proces.

2.5 Śledzenie procesów

Śledzenie procesów jest konieczne do sprawowania kontroli przez węzły, które stworzyły proces. Chodzi np: o zwracanie informacji o tym, że proces się zakończył i można powtórnie użyć jego pid. Przykładowo: jeśli proces A stworzony na węźle 1 po czym migruje na węzeł 2 i tam zakończy swoje działanie węzeł 1 zostanie o tym poinformowany tak aby mógł w przyszłości stworzyć proces o podanym numerze pid.

Tak więc podczas kończenie procesu A węzeł 2 wyśle odpowiednią informację do węzła 1 (do którego węzła ma wysłać wie przez mapowanie numeru pid na numer węzła). Wtedy węzeł 1 będzie ewentualnie mógł przystąpić do pewnych działań (np: związanych z pokrewieństwem czy grupami procesów).

2.6 Przechowywanie struktur vproc i pvproc

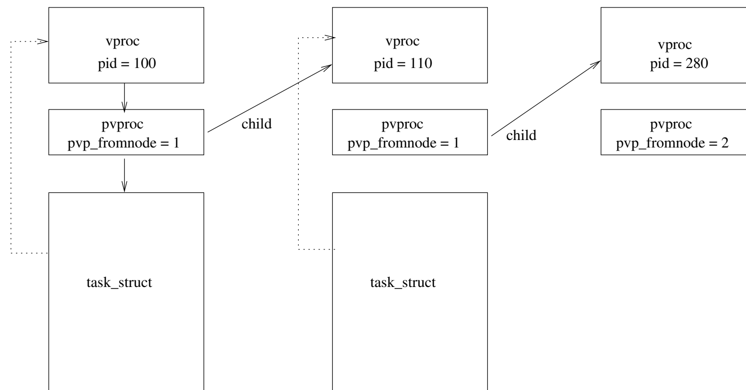
Struktura *task_struct* procesu obecna jest tylko na węźle, który wykonuje proces. Natomiast struktury *vproc* i *pvproc* dostępne są tylko na następujących węzłach:

- Na *local node* (węźle, na którym wykonuje się proces)
- Na *origin node* (węźle, na którym stworzono proces)
- Na węźle, na którym wykonują się dzieci podanego procesu
- Na węźle, na którym wykonuje się rodzic podanego procesu
- Na każdym węźle, który wykonuje proces z grupy procesów będzie struktura proc/vproc dla lidera tej grupy
- Na każdym węźle, który wykonuje proces będący liderem grupy należącej do pewnej sesji będzie struktura proc/vproc lidera tej sesji
- Tymczasowe struktury proc/vproc np: podczas wysyłania do podanego procesu sygnału, albo w trakcie wydawania polecenia ps (dla wyświetlenia informacji o wszystkich procesach)

Należy dodać, że struktury *vproc* i *pvproc* są przechowywane tylko tam gdzie są potrzebne. Tymczasowo mogą pojawiać się na innych węzłach, jednak zostają usuwane tak szybko jak to możliwe. Ponadto cały system działa tak aby maksymalnie opóźniać przesyłanie danych, które aktualizują stan tych struktur. Oznacza to, że jeśli nie jest to absolutnie potrzebne w naszym kernelu mogą być nieaktualne wartości tych struktur i będą one aktualizowane tylko wtedy kiedy będzie to potrzebne (np: po wydaniu polecenia ps). Dopiero wtedy nastąpi przesłanie poprzez sieć (na poziomie kernela) aktualnych danych. Jest kilka wyjątków, dla których jest wymagana natychmiastowa aktualizacja pewnych danych. W celach optymalizacji rodzic powinien znać bieżący stan i flagi swoich dzieci. Dlatego węzeł, który wykonuje rodzica w pewnych polach struktury *pvproc* opisujących jego dzieci posiada zawsze aktualne dane (aktualizowane na bieżąco). Takim polem jest np: *pvp_wstate*.

2.7 Rozpraszanie procesów

Węzeł na którym wykonuje się aktualnie proces, zwany jego węzłem lokalnym przechowuje wszystkie trzy struktury procesu: *vproc*, *pvproc* oraz *task_struct*. Jeśli proces, który został na nim stworzony aktualnie wykonuje się na innym węźle *origin node* posiada jedynie struktury *vproc* i *pvproc* dla tego procesu. Dzięki ogólnodostępnemu systemowi /proc każdy węzeł klastra może posiąść informacje dotyczące tego gdzie dany proces się wykonuje i jakie są jego relacje rodzicielskie. To na podstawie relacji rodzicielskich zawartych w *pvproc* działa system klastrowy. Jednak musi on zmieniać również wpisy dotyczące tych relacji w *task_struct* tak, aby i one miały poprawne dane (mogą być wykorzystane zarówno w kernelu jak i przez programy użytkowe poprzez wywołania systemowe).



Rysunek 3: Struktury pamięci dla węzła nr. 1

Należy pamiętać, że przeglądanie całej listy procesów rozproszonych nie jest to tak dobrze zoptymalizowane jak przeglądanie listy procesów rzeczywistych (tych, które się wykonują na naszym węźle, będących w *task_struct*). Wiąże się to z tym, że dla każdego węzła należy wysłać żądanie zwrócenia informacji o procesach, dla których jest on *origin_node*. To właśnie węzeł, który oryginalnie stworzył proces kontroluje tenże proces - mimo, że może on nie posiadać *task_struct* dla podanego procesu, co może podczas przeglądania listy procesów wiązać się z komunikacją z węzłem na którym proces się aktualnie wykonuje (np: w celu pobrania szczegółowych informacji z *task_struct*).

Ta sieciowa komunikacja na poziomie kernela jest dość dobrze zoptymalizowana, jednak zawsze są jakieś narzuty z nią związane i przy dużej ilości migrujących procesów (dużej dynamice) wymagają szybkiej sieci lokalnej do sprawnego działania klastra.

Na rysunku widać, że gdy proces nie wykonuje się na naszym węźle to nie posiadamy jego *task_struct*. Gdyby proces o pid=280 nie był dzieckiem procesu pid=110 to nie posiadalibyśmy nawet jego struktur *vproc* i *pvproc*. Nic byśmy o nim nie wiedzieli dopóki nie zadalibyśmy odpowiedniego zapytania (np: ps ax), kiedy to kernel wysłałby komunikaty do pozostałych węzłów i stworzył tymczasowe struktury *vproc* i *pvproc* dla odpowiednich procesów.

2.8 Migracja procesów

Proces migracji zawsze jest kontrolowany przez *origin_node* (węzeł, który stworzył dany proces). Jeśli proces X został stworzony na węźle A, a teraz pracuje na B i chce migrować na C to musi powiadomić o tym fakcie tylko węzeł A (no i oczywiście węzeł C, na który się przenosi). *origin_node* jest znajdywany na podstawie mapowania pid na numery węzłów. Należy zauważyć, że dane w *vproc* i *pvproc* na innych węzłach mogą być już nieaktualne.

W praktyce program użytkowy migrate przenoszący proces na inny węzeł po wywołaniu np: migrate 600 2 (przenieś proces 600 na węzeł 2) otwiera plik /proc/600/goto i zapisuje tam 2. Wtedy odpowiednia procedura obsługi w kernelu (zarejestrowana dla tego pliku) kontaktuje się z węzłem *origin node* dla danego procesu i powiadamia go o tym fakcie (ustawia pole *pvp_tonode* w *pvproc*, które jest również ustawiane na

węźle, z którego proces migruje).

Na węźle, z którego przenosi się proces podczas powrotu z wywołania systemowego (lub z przerwania) do przestrzeni użytkownika sprawdzane jest pole *pvp_tonode* i jeśli jest ono różne od *pvp_fromnode* następuje właściwy etap migracji, po zakończeniu którego proces powraca do przestrzeni użytkownika na nowym węźle. Właściwy etap migracji składa się z:

- Kopiowaniu części danych procesu (z *task_struct*) do nowego węzła (oraz ich odpowiednie zmiany np: pola *pvp_fromnode*)
- Kopiowania brudnych stron do nowego węzła
- Ponowne otwieranie plików (urządzeń i obiektów IPC)
- Innymi opcjonalnymi czynnościami (np: ponowne otwarcie obiektów shm jeśli trzeba)

Na koniec następuje aktualizacja struktury *pvproc* w *origin node*, które pole *pvp_frommode* od tej chwili wskazuje juz na nowy węzeł.

Dla użytkownika dostępne jest również wywołanie systemowe migrate więc program użytkowy migrate wcale nie musiałby korzystać z /proc, jednak idea działania jest taka sama.

2.9 Migracja procesów - dynamika

W trakcie działania klastra węzły mogą zostać czasowo (albo na stałe) odłączone, co prowadzi do szeregu komplikacji związanych z relacjami rodzicielskimi między procesami. Podobne komplikacje związane są z węzłem, który stworzył proces (*origin node*) a który jest odpowiedzialny za jego migrację w momencie, gdy zostanie on odłączony do klastra.

Jeśli proces pracuje na węźle B a został stworzony na A i węzeł A zostanie odłączony, wtedy poprzez odpowiedni algorytm zostaje wybrany dla niego zastępczy węzeł (*surrogate origin node*) np: węzeł C. Każdy węzeł ma zawsze aktualne struktury opisujące jakie węzły są aktualnie w sieci. Jeśli zauważy, że dany węzeł się odłączył mapuje odpowiednie numery pid (z przestrzeni węzła, który się odłączył) na kolejny pracujący węzeł, który dla tych procesów teraz staje się (*surrogate origin node*). Jeśli na *origin node* (węzeł A) pracował rodzic dziecka pracującego na węźle B i węzeł A został odłączony musi jednocześnie nastąpić zmiana relacji rodzicielskich, tak aby rodzicem dziecka pracującego na węźle B był proces init (należy zauważyć, że teraz *pvp_ppid* jest różne od *pvp_oppid* dla tego dziecka). Po powrocie węzła A odpowiednie struktury wrócą do poprzedniego stanu, co tyczy się również relacji rodzicielskich (gdyby tak nie było to proces rodzic z węzła A wskazywał by na dziecko, które twierdziłoby, że jego rodzicem jest proces init).

Podobne zmiany są potrzebne podczas gdy np: na odłączanym węźle będzie chodził lider grupy procesów (w takim przypadku nie można pozostawić grupy bez lidera, a i nie można pozabijać procesów w grupie). Wbrew przypuszczeniom obsługa tego typów problemów nie jest banalna i zajmuje sporą część kodu kernela openssi.

2.10 API dla użytkownika

2.10.1 Wywołania systemowe

W standardowym kernelu 2.4.29 zdefiniowano 252 wywołania systemowe, tutaj w zmodyfikowanej wersji kernela 2.4 jest ich 294. Argumenty do omawianych wywołań przekazywane są przez rejestry (*pt_regs*). Najważniejsze wywołania związane z klastrem to:

- *ssisys* Jedno z podstawowych wywołań systemowych klastra. Po przekazaniu sterowania z sys_ssisys() do do_ssisys() następuje wykonanie jednej z wielu operacji (numer operacji i argumenty pakowane w strukturze *ssisys_iovec_t*). Operacje te wiążą się głównie z konfiguracją klastra (rozbijaną na kilka etapów takich jak PREROOT, POSTROOT, INITPROC czy MOUNT).
- *rfork* Odpowiedzialne za tworzenie nowego procesu potomnego na wskazanym węźle. Jedynym argumentem tego wywołania jest numer docelowego węzła. Funkcja sys_rfork() wywołuje dvp_rfork(), która wykonuje całą pracę.
- *rexecve* Odpowiadające wywołaniu exec(), tyle, że jako czwarty argument pobiera numer węzła, na którym ma zostać wykonane. Funkcja sys_rexecve() wywołuje dvp_rexecve(), która wykonuje całą pracę.
- *migrate* Odpowiadający za migrację procesów. Jedynym argumentem jest numer docelowego węzła. Funkcja sys_migrate() wywołuje dvp_migrate(), która wykonuje całą pracę.

Jak dla wywołania execve() tak i dla rexecve() istnieje szereg podobnych funkcji w bibliotece libcluster.so (rexecl, rexecle, rexeclp, rexecv, rexecvp, rexecve), które opakowują wywołanie rexecve(). We wspomnianej bibliotece są również odpowiednie funkcje opakowujące powyższe wywołania systemowe.

2.10.2 Proc

Dla użytkownika dostępny jest również rozbudowany system plików proc poprzez który w łatwy sposób można dokonywać zmian w konfiguracji samego klastra jak i poszczególnych węzłów i procesów wchodzących w jego skład. Analizując programy użytkowe można odnieść wrażenie, że jest to preferowany (nad wywołaniami systemowymi) interfejs programistyczny.

Tak jak w tradycyjnym systemie proc w katalogu głównym znajduje się szereg katalogów i plików związanych z fizycznym sprzętem, na którym pracuje podany system. Jednak należy pamiętać, że katalogi z numerami procesów są wspólne dla całego systemu (co nie oznacza, że każdy węzeł widzi w każdej chwili wszystkie i aktualne katalogi procesów). Dodatkowo w /proc zostały stworzone następujące wpisy (*entries*) poprzez, które można kontrolować zachowanie klastra (tylko ważniejsze wpisy):

- *cluster/node{N}* - kontroluje zachowanie węzła o numerze N. Dostępne są pliki:
  - *load* - obciążenie węzła (wg. MOSIX)
  - *loadlevel* - czy włączono równoważenie obciążenia (1 to tak)
- *cluster/loadlevellist* - lista aplikacji, które mogą automatycznie migrować

- *cluster/events* - informacje o przyłączeniach, odłączeniach węzłów
- *PID* - PID to numer procesu
  - *where* - na którym węźle się proces wykonuje
  - *pin* - przywiązanie procesu do węzła, na którym się wykonuje
  - *loadlevel* - czy włączono równoważenie obciążenia dla tego procesu(1 to tak)
  - *goto* - używany do migracji procesów (zapisuje się numer węzła, na który proces ma migrować)

2.10.3 Funkcje biblioteczne

Funkcje te dość często korzystają z wcześniej wspomnianych (i tych nie wspomnianych) wywołań systemowych oraz danych z /proc. Znajdują się w bibliotece libcluster.so. Ważniejsze z nich to:

- *node_pid()* - zwraca numer węzła na którym wykonuje się podany proces
- *clusternode_setinfo()* - ustawia status węzła
- *clusternode_num()* - zwraca numer węzła
- *clusternode_info()* - zwraca szczegółowe informacje o węźle
- *clusternode_get_ip()* - zwraca adres ip węzła
- *clusternode_avail()* - testuje czy podany węzeł jest dostępny
- *cluster_transition()* - zwraca informacje o przyłączeniach i odłączeniach od klastra począwszy od pewnego przekazanego jako parametr numeru (transid)
- *cluster_detailedtransition()* - zwraca szczegółowe informacje dotyczące ostatnich zdarzeń na podstawie numeru transid
- *cluster_ssiconfig()* - testuje czy środowisko openssi jest dostępne
- *cluster_name()* - zwraca nazwę klastra
- *cluster_membership()* - zwraca numery dostępnych węzłów klastra
- *cluster_maxnodes()* - zwraca maksymalną liczbę węzłów w klastrze
- *cluster_getnodebyname()* - zwraca numer węzła na podstawie nazwy
- *cluster_getnodebyname()* - zwraca numer węzła na podstawie nazwy
- *cluster_events_register_signal()* - pozwala aplikacjom rejestrować sygnały rejestrowane przez klaster (np: o odłączaniu, dołączaniu węzłów)

2.10.4 Przykłady wykorzystania API

Zaprezentowane zostały niewielkie wycinki kodu obrazujące wykorzystanie API o którym pisałem. Jednocześnie chciałem aby rozrzeszały one funkjonalność projektu openssi (a nie ją powielały). Na programy te składają się:

- *infocluster* - pokazuje szczegółowe informacje na temat transakcji w klastrze.
- *infonode* - pokazuje szczegółowe informacje na temat podanego węzła.
- *migrate_t1* - test szybkości migracji procesów przy użyciu systemu /proc.
- *migrate_t2* - test szybkości migracji procesów przy użyciu wywołania migrate().
- *rexec_t1* - test szybkości zdalnego wykonywania poleceń na podanym węźle.

Do programu migrate_t1 przekazuje się liczbę migracji, numer procesu, który ma migrować oraz opcję związaną z debugowaniem. Wskazany proces będzie migrował pomiędzy dostępnymi węzłami klastra (w systemie round-robin). Po zakończeniu tego procesu wyświetlone zostają statystyki jak długo trwał ów proces. Program migrate_t2 różnie się tym od migrate_t1, że wykorzystuje wywołanie systemowe migrate(), które działa zawsze na bierzącym procesie (nie można podać numeru procesu, który ma być migrowany). Program rexec_t1 pobiera numer węzła, ilość wywołań programu, ścieżke do wywoływanego programu i flagi związane z debugowaniem. Jego działanie polega na wywoływaniu fork() po czym wykonywaniu rexecv() na podanym węźle. Dzięki temu można sprawdzić jak szybko wykonują się różne programy na różnych węzłach.

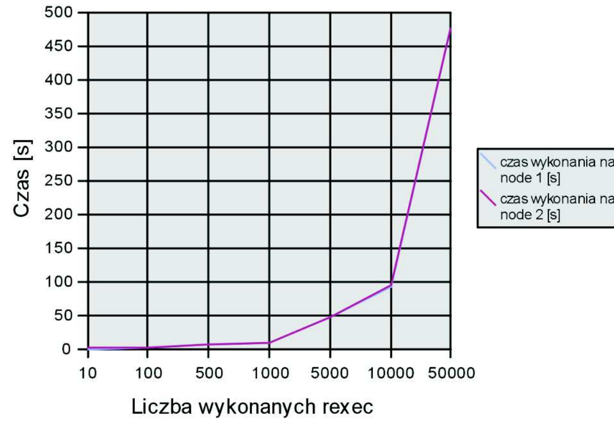
Oprócz powyższych programów zamieściłem kilka przykładowych programów z projektu openssi, które są bardzo łatwe do analizy i mogą być pomocne przy analizowaniu API. Na programy te składają się:

- *loadlevel* - słuzy do sterowania równoważenia obciążenia
- *loads* - pokazuje obciążenia węzłów
- *migrate* - służy do migracji procesów
- *onall* - wykonuje program na każdym węźle
- *onnode* - wykonuje program na podanym węźle
- *where_pid* - pokazuja na jakim węźle wykonuje się podany proces

2.11 Testy wydajnościowe

Testy porównujące wydajność systemu dla różnej liczby węzłów bazują na analizie wyników programów migrate_t1, migrate_t2 i rexec_t1.

rexec_t1	Liczba wykonanych rexec						
	10	100	500	1000	5000	10000	50000
czas wykonania na node 1 [s]	0,46	1,77	5,73	9,72	47,51	93,28	475,73
czas wykonania na node 2 [s]	1,71	1,97	5,84	9,91	46,23	95,12	474,45

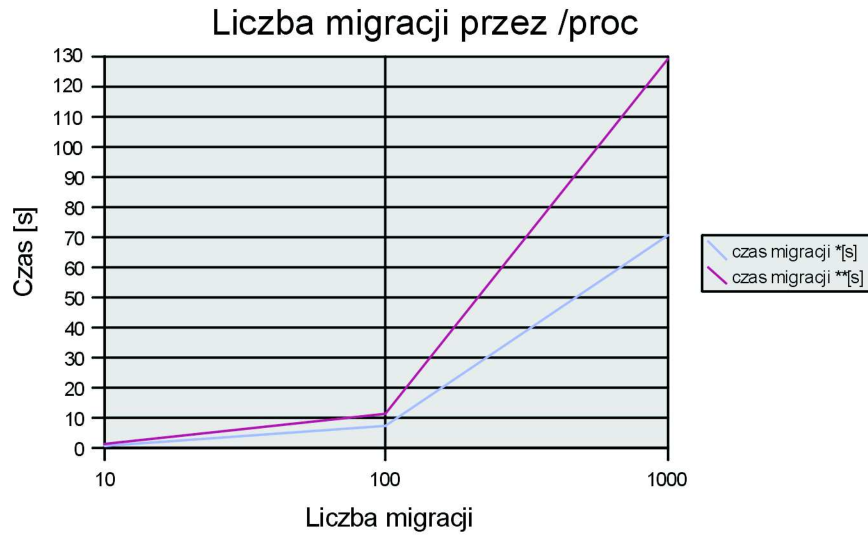


Rysunek 4: Testy rexec_t1

	Liczba migracji programu		
migrate_t1 (przez /proc)	10	100	1000
czas migracji *[s]	0,66	7,32	70,73
czas migracji **[s]	1,37	11,37	129,35

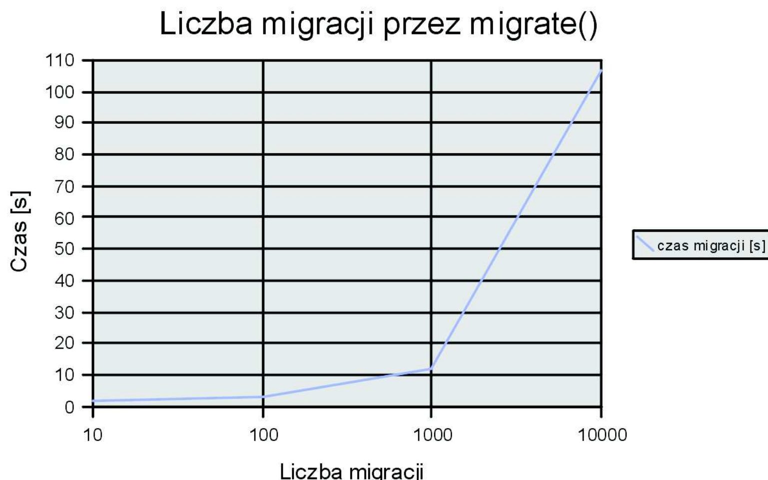
* dla procesu z aktualnego noda

** dla procesu ze zdalnego noda



Rysunek 5: Testy migrate_t1

	Liczba migracji programu			
migrate_t2 (przez migrate())	10	100	1000	10000
czas migracji [s]	1,61	2,94	11,7	106,87



Rysunek 6: Testy migrate_t2

Jak widać czas zarówno migracji jak i zdalnego wykonywania rośnie liniowo zraz z liczbą wykonywanych operacji. Średnio jedna operacja zdalnego wykonania rexec(jednostkowy program dopisywał do pewnego pliku jeden bajt) trwała 10ms. Średnio jedna migracja przy użyciu /proc trwa około 100ms, natomiast przy użyciu prostszego wywołania migrate trwa to tylko około 10ms. Jak widać za wygode i duże możliwości /proc musimy płacić wydajnością.
Testy potwierdziły również, że nie wszystkie procesy da się migrować. W szczególności nie można tego robić z procesami które zajmują pewne zasoby związane np: z terminalem (jak proces midnight commandera). Pełna lista procesów, których migracja jest nie możliwa dostępna jest na systemowej stronie podręcznika (man migrate). Jednocześnie zauważyć tu można słabość API systemu /proc, który w żaden sposób nie poinformuje o nieudanej próbie migracji w takim przypadku.
Ciekawe wydaje się porównanie migracji z wykorzystaniem /proc dla dwóch rodzajów procesów: procesu, który początkowo znajduje się na bieżącym węźle (to jego origin_node), oraz procesu, który początkowo znajduje się na innym węźle (jego origin_node nie jest bieżącym węzłem). Widać wyraźnie, że prędkość wykonywania takich migracji jest niemal dwukrotnie większa dla procesu, którego origin_node jest bieżącym węzłem. Niestety ujawnia to niedoskonałości openssi, podkreślając fakt, że każdy proces jest zarządzany z origin_node, i każda migracja procesu (oraz inne ważniejsze zmiany) wiąże się z komunikacją ze wspomniamym węzłem zarządzającym, mimo, że sam proces może w tym czasie migrować między zupełnie innymi węzłami.

Literatura

[DB02] Marco Cesati Daniel Boviet. *Understanding The Linux Kernel*. Wydawnictwo RM (O'Reilly and Associates, Inc.), 2002.

[Rub99] Alessandro Rubini. *Linux Device Drivers*. Wydawnictwo RM (O'Reilly and Associates, Inc.), 1999.

[Ver01] Michael Beck Harald Bohme Mirki Dziadzka Ulrich Kunitz Robert Magnus Dirk Verworner. *Linux Kernel Internals*. Wydawnictwo MIKOM (Addison-Wesley), 2001.

[wwwa] www.openssi.org. *Grupa dyskusyjna ssic-linux-devel@lists.sourceforge.net*.

[wwwb] www.openssi.org. *Kody źródłowe i dokumentacja projektu*. cvs -z3 -d:pserver:anonymous@cvs.openssi.org/cvsroot/ci-linux co ci.