DistLinux Projekt

Idee und Prototyp

Volkmar Sieh 2/2000

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In allen größeren Einrichtungen, in nahezu jeder Firma stehen eine Reihe von Rechnern den Mitarbeitern zur Verfügung. Die jeweiligen Systemverwalter (z.T. sogar die Benutzern selbst) tragen die Verantwortung, dass die Systeme ,,vernünftig" installiert sind. Jeder Rechner wird für sich installiert und gewartet. Ergibt sich die Notwendigkeit, Software zu updaten, so muss dies auf allen Systemen separat geschehen. Probleme ergeben sich dann, wenn einzelne Rechner kurzfristig nicht verfügbar sind (defekt, abgeschaltet). In diesen Fällen wird häufig vergessen, die notwendigen Updates später nachzuholen. Das Problem klingt trivial. In der Praxis sind die auf diese Weise sich verschieden entwickelnden Systeme jedoch eines der größten Probleme der Systemverwaltung.

Bedingt durch die Tatsache, dass jede(r) Mitarbeiter(in) (mindestens) einen Computer für ihre/seine tägliche Arbeit benötigt, steht an jedem Arbeitsplatz ein Computer bereit. Jeder dieser Rechner muss so ausgelegt sein, dass er mindestens die Rechenleistung zur Verfügung stellen kann, die der Benutzer benötigt. Beobachtet man über einen längeren Zeitraum die einzelnen Computer, so stellt man jedoch fest, dass die Rechenleistung der Systeme nur wenige Minuten am Tag wirklich ausgenutzt wird. Meist warten die Systeme auf neue Eingaben des Benutzers. Auf der anderen Seite gibt es jedoch häufig auch Aufgaben, die sehr viel Rechenzeit über mehrere Stunden hinweg brauchen. In diesen Fällen wäre es schön, die überzählige Rechenkapazität der anderen Arbeitsplätze mitnutzen zu können.

Um hohe eine Rechenleistungen zu bekommen, werden daher häufig Methoden verwendet, die auf mehreren Rechnern Programme starten, die dann über Nachrichten über das Netz miteinander kommunizieren und ihre Ergebnisse austauschen. Häufig ist bei dieser Art der Programmierung weniger das eigentlich zu lösende Problem schwierig zu implementieren. Viel aufwändiger dagegen ist die Kommunikation zwischen den einzelnen Rechnern, die Verteilung der Teilaufgaben und das Zusammentragen der Einzelergebnisse sowie die Koordinierung der Rechner zu lösen. Einfacher ist die Programmierung von Multiprozessorsystem, die zumindest über einen gemeinsamen Speicher und eine gemeinsame Uhr verfügen. Leider lassen sich aber weder Multiprozessor-Programme auf einem Workstation-Cluster noch Workstation-Cluster-Programme auf einem Multiprozessorsystem (effizient) ausführen. Je nach zur Verfügung stehender Hardware ist ein vollständig anderes Programm zu entwickeln.

Das Ziel sollte es daher sein, alle Rechner so miteinander zu vernet-

zen, dass sie nach aussen hin (für Benutzer *und* für Programmierer) wie ein einzelner Rechner wirken, zusammen aber eine große Rechenleistung bieten können. Trotz der Zusammenschaltung muss es aber natürlich möglich sein, Teile des Systems (einzelne Rechner) herunterfahren zu können (z.B., um sie mit neuer Hardware aufzurüsten). Ebenso sollten neue bzw. bisher aussenstehende Rechner jederzeit in das Gesamtsystem integriert werden können.

Das im folgenden beschriebene **DistLinux**-Projekt zeigt neue Ideen, wie ein solches System aufgebaut werden könnte. Im nachfolgenden Kapitel werden die Ideen selbst beschrieben. Die Umsetzung der aufgezeigten Ideen in einem Prototyp wurde durchgeführt. Der Aufbau des Prototyps wird im Kapitel darauf vorgestellt.

2 Idee

2.1 `migrate`-System-Call

Um den Umstieg auf ein ,,vernetztes" System zu vereinfachen, sollte sich an der Schnittstelle, die das Betriebssystem bietet, möglichst wenig ändern. Nur dann ist es möglich, viele der bestehenden Anwendungen auf ein solches System direkt zu übernehmen bzw. einfach zu portieren. Viele der bestehenden verteilten System kranken daran, dass es keine Software für sie gibt.

Daher entstand die Idee, die Systemschnittstelle (in diesem Fall Linux Version 2.2) *unverändert* zu übernehmen und sie nur um einen einzelnen System-Call zu erweitern. Der neu hinzukommenden Systemaufruf `migrate` ermöglicht es dem aufrufenden Prozess, von einem Knoten auf einen anderen zu migrieren. Benötigte Betriebsmittel (Speicherseiten, Dateien, usw.) werden ,,on-demand" nachgeschickt.

Es erfolgt *kein* automatisches Load-Balancing über die `migrate`- Aufrufe. Statt dessen rechnet jeder Prozess auf seinem, von ihm benannten ,,Wunschknoten". Der `migrate`-System-Call stellt einen Mechanismus dar, mit Hilfe dessen Prozesse verschoben werden können. Die Strategien dazu können u.U. darauf aufsetzend auf User-Ebene implementiert werden.

Aus Sicht der Programmierer kann der `migrate`-Aufruf wie eine Optimierungsanweisung angesehen werden (ähnlich der `register`- Anweisung in einem C-Programm). Mit dieser Anweisung kann ein Programmierer angeben, wie sein Programm vermutlich am schnellsten abgearbeitet wird. Unabhängig von der Ausführung dieser Anweisung wird das Ergebnis des Programmlaufes jedoch – bis auf Zeitaspekte – immer dasselbe sein. D.h. man kann für Debug- oder Verifikation- Aufgaben die `migrate`-Anweisungen einfach aus dem Programm- Code streichen. Damit vereinfachen sich z.B. Verifikationsaufgaben enorm, da auf diese Weise aus einem verteilten, parallelen Programm ein normales, paralleles Programm wird.

2.2 Beispiele

Um die große Flexibilität des `migrate`-System-Calls zu zeigen, soll an den folgenden Beispielen verdeutlicht werden, auf welche Weise bekannte Programmierungsparadigmen auf den `migrate`-System-Call (und weitere bereits vorhandene Unix-System-Calls) abgebildet werden können.

2.2.1 Remote Procedure Calls

Mit Hilfe des `migrate`-System-Calls können auf einfache Weise Remote-Procedure-Calls (RPCs) nachgebildet werden. Da auf alle Daten von allen Knoten aus zugegriffen werden kann, entfällt das bei RPCs sonst übliche, mühsam zu programmierende Argument- und Result-Marschalling. Zusätzlich ist es möglich, Pointer als Argumente zu übergeben. Auch ist es nicht notwendig, auf der Server-Seite einen Prozess zu starten, der die Argumente entgegennimmt, die Funktion ausführt und Resultate zurückschickt. Statt dessen kann man den aufrufenden Prozess selbst zur Ausführung der Funktion verwenden.

Das folgende Beispiel emuliert einen RPC, der eine Datenbank abfragt und modifiziert.

```
...
client = migrate(server);

sem_lock(database);
... = database[...]->...;
database[...]->... = ...;
sem_unlock(database);

(void) migrate(client);
...
```

2.2.2 Message-Passing

Im nachfolgenden Beispiel wird gezeigt, wie ein Prozesssystem, das über Nachrichten miteinander kommuniziert, nachgebildet werden kann. Gezeigt werden die Sende- bzw. Empfangsroutine. Nachrichten werden über Shared-Memory-Bereiche übertragen. Die Koordinierung des Zugriffes auf die Bereiche erfolgt über Semaphoren (`lock_mailbox` bzw. `unlock_mailbox`). Die Synchronisation kann über Signale erfolgen (`block_sigs`, `unblock_sigs`, `wait_for_signal` bzw. `kill`).

```
send(pid_t receiver, char *msg, int msgsize) {
    lock_mailbox(receiver);
    add_to_mailbox(receiver, msg, msgsize);
    unlock_mailbox(receiver);
    kill(receiver, SIGUSR1);
}
```

```
recv(pid_t *sender, char **msg, int *msgsize) {
    pid_t myself = getpid();

    block_sigs();
    lock_mailbox(myself);
    while (mailbox_empty(myself)) {
        unlock_mailbox(myself);
        wait_for_signal(SIGUSR1);
        lock_mailbox(myself);
    }
    unlock_mailbox(myself);
    unblock_sigs();
    lock_mailbox(myself);
    remove_from_mailbox(sender, msg, msgsize);
    unlock_mailbox(myself);
}
```

2.2.3 Shared-Memory

Wird eine Kommunikation über Shared-Memory bevorzugt, ist dies auf diesem angedachten System ohne großen Aufwand ebenfalls möglich. Dies wird am folgenden Beispiel gezeigt:

```
/* allocate shared memory */
mem = shm_alloc(size);

/* initialize shared memory */
mem->... = ...;

/* create processes */
if (fork() == 0) { /* child */
    /* migrate to different host */
    migrate(host1);

    /* use shared memory */
    ... = mem->...;
    mem->... = ...;

} else { /* parent */
    /* migrate to different host */
    migrate(host2);
```

```
    /* use shared memory */
    ... = mem->...;
    mem->... = ...;
}

/* wait for child */
wait(NULL);

/* free shared memory */
shm_free(mem);
```

Selbstverständlich können zur Koordinierung bzw. Synchronisierung zusätzlich weitere Standard-Unix-Mechanismen (z.B. Pipes oder Signale) eingesetzt werden.

2.3 Hints

Wandern Prozesse zwischen verschiedenen Knoten hin und her, so müssen sie jeweils ihre Umgebung (z.B. geöffnete Dateien, Speicherbereiche, Current-Working-Directory, usw.) mitnehmen. So soll beispielsweise das folgende Code-Fragment korrekt verarbeitet werden, egal, ob die `migrate`-Anweisungen ausgeführt werden oder nicht. Die `read`- bzw. `write`-Operationen sollen immer auf die gleiche (eventuell auf einem anderen Knoten liegende) Datei wirken.

```
/* open file */
fd = open(filename, O_RDWR);

if (...) migrate(otherhost);

/* use file */
read(fd, buffer, sizeof(buffer);
write(fd, buffer, sizeof(buffer);

if (...) migrate(otherhost);

/* close file */
close(fd);
```

Dabei ergibt sich das Problem, dass immer wieder Objekte im Gesamtsystem wiedergefunden werden müssen. Im obigen Beispiel muss auf die geöffnete Datei jederzeit zugegriffen werden können. Dies ist am einfachsten über

Broadcasts möglich. Um z.B. im obigen Beispiel in die geöffnete Datei etwas zu schreiben, könnte der Prozess in einer Broadcast-Nachricht an alle Knoten die notwendigen Daten verschicken. Der Knoten, auf dem die Datei zur Zeit liegt, kann dann die notwendige Operation durchführen. Um die Kommunikations-Hardware jedoch nicht unnötig zu belasten, sollten Broadcasts auf dem Netz wenn möglich vermieden werden. Sonst skalliert ein solches System sehr schlecht und ist nur für einige wenige Knoten vernünftig einsetzbar.

Zur Lösung dieses Problems wurde bereits früher [1] sogenannte Hints vorgeschlagen. Zu jedem Objekt speichert jeder Knoten einen Hinweis ab, wo das Objekt (vermutlich) zu finden ist. Fordert ein Knoten ein Objekt an, können alle Knoten, die dies mitbekommen (weil sie nach dem Objekt gefragt wurden), ihre Hints entsprechend anpassen. Auf diese Weise entstehen Ketten von Verweisen, die letztendlich zum gesuchten Objekt führen.

Diese Lösung erscheint zunächst einleuchtend. Sie hat jedoch entscheidende Probleme:

- Auf jedem Knoten werden sich sehr viele ($> 10^5$) Objekte befinden. Besteht der Cluster aus z.B. 256 Knoten, so müssen damit für $256 * 10^5$ Objekte auf jedem Knoten Hints verwaltet werden. Daraus ergibt sich – allein für der Verwaltung der Objekte – ein Speicherbedarf von 25 MByte pro Knoten bzw. ca. 6,5 GByte für den Gesamt-Cluster.

- Im Gegensatz zum Ansatz in [1] sollen Objekte erzeugt und auch während der Laufzeit des Systems wieder gelöscht werden können. Man kann sich leicht überlegen, dass damit ein noch größerer Speicherbedarf notwendig ist, weil Hint-Ketten noch gespeichert werden, die auf nicht mehr existierende Objekte verweisen. Zum anderen existieren natürlich keine Hint-Ketten zu gerade erst erzeugten Objekten.

- Ein großer Vorteil des beschriebenen Systems ist, dass während der Laufzeit des Gesamt-Clusters einzelne Knoten heruntergefahren bzw. gebootet werden können. Knoten, die gerade erst gebootet haben, kennen aber natürlich keinerlei Hints und müssen demnach über andere Verfahren an die von ihnen benötigten Objekte kommen. Werden Knoten abgeschaltet, können Hint-Ketten unterbrochen werden. Auch in diesem Fall sind zusätzliche Methoden notwendig, um Objekte wieder auffinden zu können.

Um diese Probleme zu lösen, wurde der im folgenden beschriebene Algorithmus entwickelt.

Auf jedem Knoten werden nur Hints auf alle lokalen Objekte und auf die N zuletzt referenzierten Objekte (mit Time-Stamp der letzten Referenzierung) gespeichert. Wird ein Hint für ein Objekt benötigt, der nicht in der Liste enthalten ist, wird über einen Broadcast von allen Knoten der Hint angefordert. Jeder Knoten antwortet darauf mit seinem Hint (sofern gespeichert) und dem dazugehörigen Time-Stamp. Hier können mehrere verschiedene Fälle eintreten:

- Im ersten Fall besitzt genau ein Knoten das Objekt. Dieser Knoten wird dann den neuesten Hint zurücksenden. Dieser kann verwendet werden, um das Objekt zu holen.
- Das Objekt wird gerade vom Knoten A zum Knoten B verschoben. Dann zeigt der Hint auf dem Knoten A auf den Knoten B und der Hint auf dem Knoten B auf den Knoten A. In diesem Fall kann jeder der beiden Hints verwendet werden. Beide führen direkt bzw. einfach indirekt zum gesuchten Objekt.
- Ist das Objekt vor kurzem gelöscht worden, werden u.U. noch Hints nach der Broadcast-Anfrage zurückgeliefert. Führt der jüngste Hint jedoch nicht zum Objekt, existiert dieses Objekt nicht mehr.
- Liefert keiner der angesprochenen Knoten einen Hint zurück, existiert das Objekt nicht (mehr).

Mit Hilfe dieses Algorithmus ist es möglich, eine große Anzahl von Objekten auf einer großen Zahl von Knoten effizient wiederzufinden. Objekte, die erst vor kurzem verwendet wurden, können ohne Broadcast aufgefunden werden. Nur bei seit langem unbenutzten bzw. neuen Objekten muss mit Hilfe von Broadcasts auf allen Knoten nachgefragt werden.

Der für die Hints zusätzlich benötigte Speicherplatz ist gering. Es müssen zusätzlich zu den lokal vorhandenen Objekten lediglich N Hints gespeichert werden. D.h. der Speicherbedarf ist unabhängig von der Anzahl der am Workstation-Cluster beteiligten Knoten.

2.4 Vor- und Nachteile

Angenommen, es wäre möglich, ein solches Betriebssystem zu entwickeln. Dann wären folgende Vor- bzw. Nachteile zu erwarten.

Für die ,,normalen" Benutzer ändert sich nichts. Sie loggen sich ,,lokal" auf ihrer Maschine ein, starten auf der dort laufenden Shell / Benutzeroberfläche ihre Programme. Da diese Programme normalerweise kein `migrate` aufrufen, bleiben die Programme lokal auf dem Rechner. Daher ist für sie

während des normalen Betriebs kein Vor- aber auch kein Nachteil zu erwarten.

2.4.1 Vorteile

- Die Systemverwaltung vereinfacht sich enorm:
  - Da der Cluster wie ein einzelner Rechner wirkt, ist auch nur ein einzelner Rechner zu administieren. D.h. Updates müssen nur einmal installiert werden. Zur Zeit abgeschaltete Knoten ,,updaten" sich automatisch bei der nächsten Integration in den Cluster.
  - In einem Verbund mehrerer einzelner Rechner sind viele Dienste notwendig, um eine gewisse Zusammenarbeit der Rechner zu gewährleisten (z.B. NFS-Daemons, NIS-Passwort-Server, Automounter, ...). Dies ist im Falle *eines* Rechners alles nicht notwendig (z.B. kann einfach eine `/etc/passwd`-Datei statt eines NIS-Passwort-Servers verwendet werden).
  - Da jederzeit Prozesse verschoben werden können, ist es möglich, einzelne Knoten komplett zu leeren. Befinden sich keinerlei Prozesse oder andere Objekte mehr auf einem Knoten, kann dieser – ohne den Rest des Clusters zu beeinflussen – abgeschaltet werden. Ebenso kann jederzeit ein neuer Knoten dem Cluster hinzugefügt werden. Dadurch ist es möglich, die Hardware einzelner Knoten zu warten ohne den Betrieb des Clusters generell zu stören. In aktuellen Systemen wird dagegen mindestens das Abschalten eines Servers störend bemerkbar sein.
- Die Entwicklung verteilter Applikationen vereinfacht sich:
  - Da der `migrate`-System-Call semantisch für ein Programm keine Bedeutung hat (es sollte ja egal sein, auf welchem Knoten ein Prozess läuft), können diese Anweisungen z.B. während des Debuggens aus dem Programm herausgenommen werden. Dadurch wird auf einfache Art und Weise aus einer verteilten Anwendung ein ,,simples", auf einem Rechner laufendes – und demnach einfacher zu debuggendes! – Programm.
  - Jedes Prozesssystem, das mit Hilfe von `migrate`-Anweisungen auf einem Workstation-Cluster verteilt wurde, kann jederzeit auch auf einem Mehrprozessor-System oder sogar Monoprozessor (mit entsprechenden Performance-Einbußen) laufen.

  - Viele verschiedene Programmierungsparadigmen (wie z.B. Message-Passing, Shared-Memory, RPCs) lassen sich auf den `migrate`-Mechanismus effizient abbilden. Daher können alle diese Paradigmen zur Programmierung eines solchen Systems (ohne Performance-Verlust!) eingesetzt werden.

- Alle bereits vorhandenen Applikationen sind ohne Änderungen lauffähig. Es ist nicht einmal notwendig, sie neu zu kompilieren! Bei richtiger Ausführung besteht Binary-Kompatibilität!

2.4.2 Nachteile

- Bedingt durch den erhöhten Verwaltungsaufwand innerhalb des Betriebsystemkerns ist u.U. eine leichte Performance-Einbuße zu bemerken. Wie groß diese ist, kann leider erst eine Implementierung eines Prototyps des Systems wirklich zeigen.

- Wie alle verteilten Systeme ist ein solches System anfällig gegenüber dem Ausfall von Teilkomponenten. Fällt ein einziger Knoten eines solchen Systems aus, bricht das gesamte System zusammen. Zwar lässt es sich sofort neu starten (ohne den ausgefallen Einzelrechner), die bei Systemabstürzen üblichen Daten- und Zeitverluste lassen sich jedoch nicht vermeiden. Das System muss daher besonders gut (d.h. mit besonders wendig Programmierfehlern) implementiert sein.

3 Prototyp

In diesem Kapitel soll auf den derzeitigen Stand des aufgebauten Prototyps eingegangen werden. Er wurde mit Hilfe von Studienarbeiten [2], [3] und [4] aufgebaut. Weitere Studien- bzw. Diplomarbeiten sind für die Zukunft bereits geplant.

Grundlage des aktuellen Prototyps sind die Linux-Sourcen (Version 2.2.14). In früheren Linux-Versionen fehlen entscheidende System-Calls (z.B. `sigaltstack`). Neuere Linux-Versionen existierten zu Beginn der **DistLinux**-Programmierung noch nicht. Es könnte jedoch sinnvoll sein, demnächst auf neuere Versionen umzusteigen, falls diese weniger Fehler enthalten als die bisher verwendete Version 2.2.14.

3.1 Linux on Top of Linux

Um die Programmierung und vor allem das Debuggen zu vereinfachen, wurde der gesamte Prototyp nicht als vollständiges Betriebssystem auf einer Hardware aufgebaut. Statt einer Hardware wird wiederum ein Linux-Betriebssystem verwendet (LoToL = **L**inux **o**n **T**op **o**f **L**inux). D.h. statt `in`- und `out`-Befehle an die Hardware zu schicken, werden System-Calls an das eigentliche Betriebssystem abgesetzt. Wie dies im einzelnen umgesetzt wurde, soll in den folgenden Abschnitten verdeutlicht werden.

3.1.1 Interrupts, System-Calls, Exceptions

Innerhalb eines Linux-Programms werden alle Events und Ausnahmen über Signale an den bearbeitenden Prozess mitgeteilt. Daher werden im **LoToL**-Ansatz alle Interrupts, System-Calls und Exceptions ebenfalls auf Signale abgebildet.

Für Interrupts werden die folgenden beiden Signale verwendet:

`SIGIO` Geräte verwenden dieses Signal, um der Anwendung mitzuteilen, dass z.B. neue Zeichen gelesen bzw. geschrieben werden können.

`SIGALRM` Immer, wenn der Alarm-Timer eines Prozesses abläuft, verschickt er dieses Signal an den Prozess.

Eine (De-) Blockierung von Interrupts (entspricht dem `sti` bzw. `cli`-Assembler-Aufruf auf einem Intel-Rechner) kann durch entsprechende `sigprocmask`-Befehle simuliert werden. Auch das Abfragen der Interrupt-Maske ist über diesen Befehl möglich.

System-Calls verschickt der laufende Prozess an sich selbst über das Signal `SIGUSR1`. Das Register `%edx` enthält einen Pointer auf einen Parameterblock. Über den Parameterblock werden die Funktionsnummer und die Funktionsparameter übergeben.

Auftretende Exceptions werden durch die folgenden Signale an den laufenden Prozess (das „Betriebssystem") gemeldet:

`SIGSEGV` Der Prozess hat auf eine nicht gemappte Speicherseite zugegriffen.

`SIGBUS` Der Prozess mit einem falschen Wort-Alignment auf eine Speicherzelle zugegriffen.

Über die Unix-Funktion `sigaltstack` kann zwischen einem User- und einem System-Stack umgeschaltet werden. Dies ist notwendig, um Page-Faults innerhalb des User-Stacks behandeln zu können. Beim Start des **LoToL**-Systems wird entsprechend ein Supervisor-Stack gesetzt. Alle Signal-Handler werden über den `sigaction`-Befehl initialisiert. Durch Angabe des `SA_ONSTACK`-Bits wird angezeigt, dass *alle* Signale auf dem Supervisor-Stack zu bearbeiten sind. Um den Stack sofort zu benutzen, schickt sich der Betriebssystem-Prozess nach dem Start ein `SIGUSR2`-Signal.

3.1.2 MMU

Die Nachbildung einer MMU macht die größten Schwierigkeiten, wenn man versucht, auf einem Betriebssystem ein weiteres Betriebssystem zu emulieren. Die einzigen Unix-Befehle, die entfernt an MMU-Befehle erinnern, sind die `mmap`- und `munmap`-Befehle, mit deren Hilfe es möglich ist, Teile von Dateien in den eigenen virtuellen Adressraum zu mappen.

Über den folgenden Umweg ist es möglich, Speicherseiten zu mappen: Zunächst wird eine Datei angelegt, die so groß ist wie der zu simulierende physikalische Hauptspeicher. Diese Datei wird mittels `open` geöffnet und offen gehalten. Jetzt kann mittels `unlink` der Eintrag im Dateisystem wieder gelöscht werden. Über `map`- bzw. `munmap`-Befehle können jetzt „Seiten" aus der Datei an verschiedene Stellen im virtuellen Adressraum gemappt bzw. aus dem Mapping herausgenommen werden.

Leider ist es mit diesem Verfahren sehr aufwändig, mehrere Seiten umzumappen. Dies ist jedoch bei jedem Kontext-Switch nötig. Daher sind Kontext-Switches im `DistLinux`-System auf `LoToL`-Basis noch sehr teuer.

3.1.3 Treiber

Jeder **DistLinux**-Treiber ruft Funktionen des Original-Linux-Kerns auf. So verwendet z.B. der Konsolen-Treiber `read`- und `write`-Aufrufe, um von der

Konsole Zeichen zu holen bzw. um dorthin Zeichen zu senden. Der Netzwerk-Treiber benutzt entsprechend `sendto`- und `recvfrom`-Aufrufe, um Netzwerk-Pakete zu verschicken bzw. zu empfangen.

Um nicht in den entsprechenden System-Calls zu blockieren, werden die `read`- bzw. `recvfrom`-System-Calls nur aufgerufen, wenn sichergestellt ist, dass wirklich Zeichen bzw. Pakete gelesen werden können. Dazu werden die Datei- bzw. Socket-Descriptoren nach dem Öffnen sofort auf asynchronen, nicht-blockierenden Betrieb umgestellt (siehe `man fcntl`). D.h. zum einen, dass ein Signal (`SIGIO`) an den Prozess geschickt wird, sobald Zeichen bzw. Pakete vom Treiber gelesen werden kann. Weiterhin wird damit sichergestellt, dass die System-Calls sich auf keinen Fall blockieren. Statt dessen liefern die System-Calls die Fehlermeldung `EAGAIN` bzw. `EWOULDBLOCK`.

Dies Verhalten entspricht in etwa dem Verhalten ,,normaler" Hardware. Sie schickt Interrupts, wenn neue Zeichen bzw. Pakete angekommen sind. Die externe Hardware blockiert aber niemals die CPU.

3.2 Ident-Manager

Werden neue Objekte angelegt, so muss jedes Objekt eine eindeutige Kennzeichnung bekommen, um gegebenenfalls im ganzen Cluster wiedergefunden werden zu können.

Aufgabe des Ident-Managers ist es, Zahlen zu generieren, die auf dem ganzen Cluster eindeutig sind. Diese Zahlen können dann als Kennzeichnung für die Objekte verwendet werden.

Besondere Schwierigkeiten bestehen in den nachfolgenden Anforderungen:

- Die Generierung von IDs muss sehr schnell erfolgen können. Beispielsweise werden bei einem `fork`-System-Call viele neue Objekte erzeugt, die zusammen den neuen Prozess ausmachen.

- Entsprechend muss der Ident-Manager in der Lage sein, große Mengen freigewordener IDs schnell wieder in entsprechende Freilisten eintragen zu können (z.B. bei einem `exit`-Call).

- Zum Teil können Benutzer IDs selbst vergeben (z.B. beim Anlegen von Shared-Memory-Bereichen (`shmget`), beim Anlegen von Message-Queues (`msgget`), usw.). Auch ist es notwendig, dass der `init`-Prozess die ID ,,1" erhält. Diese von aussen bestimmten IDs müssen entsprechend als ,,belegt" markiert werden können. Selbstverständlich muss vorher überprüft werden können, ob die IDs überhaupt noch ,,frei" sind.

Zur Zeit wird noch ein sehr eingeschränkter Ident-Manager verwendet. Dieser zählt einfach einen Zähler hoch und addiert noch seine Knotennummer multipliziert mit einem Faktor dazu. Der Zähler läuft aber recht schnell über. Zudem werden wieder als frei gemeldete IDs nicht wiederverwendet. Damit ist das Gesamtsystem zur Zeit nur über einen begrenzten Zeitraum hinweg lauffähig.

3.3 Objekt-Manager und Knoten-Manager

Aufgabe des Objekt-Managers in Verbindung mit dem Knoten-Manager ist es, alle Objekte in dem verteilten System speichern und gegebenenfalls auf andere Knoten migrieren zu können. Es ist sicherzustellen, dass jeweils nur ein Knoten das Objekt besitzt, wenn er Änderungen an dem Objekt vornimmt. Wollen mehrere Knoten nur Daten aus dem Objekt lesen, kann der Objekt-Manager mehrere Lese-Kopien an verschiedene Knoten herausgeben. Im Falle von bootenden bzw. haltenden Knoten muss sichergestellt sein, dass kein Objekt verlorengehen kann.

Nach dieser Vorgabe wurde von Martin Waitz [4] eine Studienarbeit angefertig. Danach besitzt der Objekt-Manager ein Interface wie folgt:

`obj_create` Mit Hilfe dieses Funktionsaufrufes können Objekte neu angelegt werden. Der Funktion muss der Typ und die ID des neu zu erstellenden Objektes übergeben werden. Sie liefert einen Pointer auf einen Speicherbereich zurück. Dieser Speicherbereich kann mit normalen Speicherbefehlen beschrieben und ausgelesen werden. Das Objekt ist bis zum Aufruf der `obj_unlock`-Funktion auf dem Rechner gelockt und damit nicht verschiebbar.

`obj_destroy` Mit dieser Funktion kann ein bestehendes Objekt gelöscht werden. Vor Aufruf dieser Funktion muss es mit der `obj_lock`-Funktion für Schreibzugriffe gelockt werden.

`obj_lock` Dieser Funktion wird neben dem Typ und der ID auch noch ein Flag übergeben, dass anzeigt, ob das bezeichnete Objekt nur gelesen oder auch modifiziert werden soll. Bei Rückkehr der Funktion bekommt der Aufrufer ähnlich der `obj_create`-Funktion einen Pointer auf einen Speicherbereich zurück, in dem das Objekt eingelagert wurde.

`obj_unlock` Hebt das Locking der `obj_create`- bzw. `obj_lock`-Funktion wieder auf.

`obj_move` Mit dieser Funktion kann ein angegebenes Objekt (z.B. auch ein Prozess) auf einen anderen Knoten verschoben werden. Im Gegensatz

zur `obj_lock`-Funktion, die jedes Objekt jeweils zum eigenen Knoten holt, kann mit der `obj_move`-Funktion ein Objekt auch vom eigenen Knoten weggeschoben werden.

`obj_migrate_to` Diese Funktion migriert den aufrufenden Prozess zum angegebenen Objekt und lockt dieses ähnlich der `obj_lock`-Funktion.

3.4 Prozess-Management

Das Original-Linux-System verwaltet alle Prozesse in einer einzelnen verketteten Liste. Diese Liste wird von vielen System-Calls verwendet. Z.B. trägt der `fork`-System-Call hier den neu erzeugten Prozess ein. Ein `wait4` löscht ihn wieder aus der Liste. Viele weitere System-Calls verwenden den Inhalt der Liste. So durchsucht der `kill`-Befehl diese Liste nach dem bzw. den zu benachrichtigenden Prozessen, ein `setsid`-Befehl testet anhand der Liste, ob die angegebene Session schon existiert. Zur Koordinierung der einzelnen Aufrufe werden entsprechende Read- bzw. Write-Locks gesetzt.

Leider kann auf einem verteilten Linux-System keine solche Gesamtliste existieren. Statt dessen werden auf allen Knoten jeweils Teile dieser Liste gespeichert sein. Damit ergeben sich jedoch zwei Schwierigkeiten:

- Nicht mehr alle Prozess-Daten liegen lokal vor. D.h., dass Statusabfragen und ähnliches u.U. über das Netz verschickt werden müssen, um Remote ausgeführt werden zu können.

- Es sind keine globalen Locks mehr vorhanden. Damit können die Prozesstabelle modifizierende Befehle (`fork` und `wait4`) mit Abfragebefehlen (z.B. `getppid`, `setsid`, usw.) nicht mehr auf einfache Art und Weise koordiniert werden.

In der Arbeit von Tim Felser [2] wurde zur Lösung dieses Problems folgender Vorschlag gemacht: Alle Prozesse werden nicht mehr in einer Liste sondern in einem Baum verwaltet. Der `init`-Prozess ist die Wurzel dieses Baumes. Jeder Prozess hat als Nachfolger seine Kind-Prozesse eingetragen. Bei Modifikationen und Leseoperationen an dieser Prozess-Baum-Struktur muss nur der jeweils benötigte Sub-Baum gelockt werden. Da viele Operationen nur auf den Kind-Prozessen agieren, muss so nur jeweils ein kleiner – und meist nur lokaler! – Teil der Prozesstabelle gelockt werden. Mit Hilfe der `obj_migrate_to`-Funktion des Objekt-Managers kann der Prozess-Baum dann soweit notwendig gescannt werden.

3.5 Verteiltes Dateisystem

Für Unix-Netze hat sich im Laufe der Zeit im wesentlichen NFS als verteiltes Dateisystem durchgesetzt. Es ist in der Lage, auf einem zentralen Server liegende Daten an alle Clients zu verteilen, so dass es für jeden Benutzer im Netz so aussieht, als wären die Daten lokal vorhanden.

Ein Problem dieses Ansatzes ist die Performance. Jeder Dateizugriff muss als Anfrage über das Netz zum Server geschickt, dort bearbeitet und die Antwort wiederum über das Netz zum Client zurückgeschickt werden. Bei diesem Verfahren ist selbst ein schnelles Netz bald ein Flaschenhals des Systems. Caching ist nur bedingt möglich, da sonst Dateiinkonsistenzen auftreten können. Ein weiteres Problem stellt der zentrale Server dar. Muss er (z.B. wegen einer anstehenden Wartung) heruntergefahren werden, kann kein Client mehr auf die Daten zugreifen.

In einem System, in dem jedoch ein Objekt-Manager für Datenkonsistenz sorgt, ist es möglich, das Dateisystem über alle Knotengrenzen hinweg zu implementieren. Jeder Knoten hat einen Teil des Gesamt-Dateisystems. Der Objekt-Manager entscheidet, welches Objekt wo gespeichert wird. Greifen mehrere Knoten gleichzeitig *lesend* auf ein Objekt (z.B. einen Datenblock oder eine Inode) zu, können sie gleichzeitig eine Kopie bekommen. Nur im Falle von Schreibzugriffen muss der Zugriff sequentialisiert werden.

Wenn man bedenkt, auf wie wenige Daten normale Benutzer in einem Unix-System schreibend zugreifen dürfen (i.a. nur auf die Daten in ihrem Home bzw. Projekt-Directory), wird klar, dass hier kaum Konflikte auftreten können. Alle anderen Daten (z.B. alle Dateien in den System-Directories `/bin`, `/usr/bin`, usw.) können nur gelesen und damit ohne Probleme als Kopie auf jedem Rechner vorhanden sein. Aufgrund dieser Tatsache kann man erkennen, dass auf allen Knoten praktisch alle Daten, die ein Benutzer dieses Knotens verwenden kann, lokal vorliegen können. Die Performance wird daher i.a. besser sein als die eines NFS-Dateisystems. Zudem besteht die Möglichkeit, System-Updates einfach durchzuführen und Knoten herunterzufahren bzw. zu booten.

Dieser Ansatz ist von Dominic Klumpner im Rahmen einer Studienarbeit implementiert worden [3].

3.6 Struktur

Alle in den obigen Abschnitten nicht weiter besprochenen Teile des Standard-Linux-Kerns können vom Original übernommen werden. D.h., dass die Struktur der *DistLinux*-Sourcen weitgehend mit der Struktur des Originals übereinstimmt. So wird jeder Linux-„Hacker" sich jederzeit leicht in den *DistLinux*-

Sourcen zurechtfinden.

Es ist jedoch zu bedenken, dass in vielen Bereichen nicht mehr auf normale Speicherbereiche sondern Objekte zugegriffen wird. Vor der Verwendung der Objekte müssen diese für den Lese- bzw. Schreibzugriff vom Objekt-Manager auf den lokalen Knoten geholt und dort gelockt werden. Damit wird z.B. aus dem Programmstück

```
asmlinkage int sys_umask(int mask)
{
    mask = xchg(&current->fs->umask, mask & S_IRWXUGO);
    return mask;
}
```

das folgende:

```
asmlinkage int sys_umask(int mask)
{
    struct fs_struct *fs;

    fs = obj_lock(OBJ_TYPE_FS_STRUCT, current->fsid, 1);
    mask = xchg(&fs->umask, mask & S_IRWXUGO);
    obj_unlock(OBJ_TYPE_FS_STRUCT, current->fsid, 1);
    return mask;
}
```

Man sieht: Pointer werden in den Datenstrukturen i.a. durch IDs ersetzt. Diese müssen vor der Verwendung zunächst mittels `obj_lock` in Pointer umgesetzt werden.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Durch die Mitarbeit einiger Studenten [2], [3], [4] konnte bisher ein System aufgebaut werden, dass sehr vielversprechend ist. Inzwischen können Prozesse beliebig zwischen verschiedenen Knoten hin- und hermigrieren. Ihre Umgebung wird auf für den Programmierer völlig transparente Weise über ein „transfer-on-demand"-Verfahren mitgenommen.

Die Programmierung verteilter Systeme auf dem *DistLinux*-Kernel ist – wie erwartet – verblüffend einfach und sicher. Die Verteilung einzelner Prozesse auf dem Gesamtsystem ist sehr einfach und wenig fehleranfällig. Dies ist ein sehr großer Vorteil dieses Ansatzes.

Die Performance des Systems ist leider noch gering. Dies liegt in weiten Bereichen jedoch einfach daran, dass noch viele Debug- und Check-Operationen im Code vorhanden sind. Nach ihrer Entfernung ist mit Sicherheit eine höhere Performance zu erwarten.

Ein weiterer Grund für die zur Zeit relativ geringe Betriebssystem-Performance ist die Verwendung eines Linux-System als „Hardware". Die Simulation der Hardware durch ein normales Linux-Betriebssystem kostet in vielen Bereichen massiv Performance. Z.B. kostet ein Assembler-Befehl, der den MMU-Segment-Pointer umsetzt, statt der normal zu erwartenden Mikrosekunden viele Systemaufrufe und damit viele Millisekunden. Leider muss der MMU-Segment-Pointer bei jedem Kontext-Wechsel umgeladen werden. Ein echtes **DistLinux**-Betriebssystem würde an dieser Stelle um mehrere Größenordnungen schneller sein als der existierende Prototyp.

Leider ist auch die Anzahl der `obj_lock`- und `obj_unlock`-Aufrufe sehr viel größer als vermutet. Selbst relativ einfache System-Calls rufen diese Funktionen vielfach auf. Eine Optimierung dieser Funktionen würde daher eine große Performance-Steigerung bewirken. Möglicherweise ist auch Hardware-Unterstützung denkbar.

In den meisten Fällen wird eine hohe Performance jedoch beim Rechnen benötigt, weniger dagegen innerhalb des Betriebsystems. Selbst in diesem einfachen Prototyp wird die Rechenkapazität der einzelnen Knoten in keiner Weise verringert. Reine Rechenprogramme erhalten die gesamte Rechenleistung des Workstation-Clusters!

Der Prototyp ist daher in hervorragender Weise in der Lage, rechenzeitintensive Programme zu parallelisieren und sie effizient auszuführen.

4.2 Ausblick

Aufgrund der positiven Erfahrungen mit dem existierenden Prototyp soll das System weiter ausgebaut werden. Fehlende Teile (z.B. ein vollständiger Ident-Manager sowie die Anbindung des Dateisystems an den Rest des Systems) müssen noch implementiert werden. Es existieren Vorstellungen, wie die Teilkomponenten herzustellen sind. Die Umsetzung wird aber natürlich einige Zeit in Anspruch nehmen.

Durch den bereits existierenden Prototypen sind Performance-Schwachstellen bekannt geworden (z.B. der Objekt-Manager). Hier soll versucht werden, den Rest des Systems so umzugestalten, dass der Objekt-Manager weniger verwendet werden muss. Zum anderen kann man natürlich versuchen, den Objekt-Manager zu optimieren.

Der gesamte Zustand des Gesamtsystems ist in einzelnen Objekten gespeichert. Alle diese Objekte werden über die Objekt-Manager der einzelnen Knoten verwaltet. Ist man in der Lage, die Objekt-Manager fehlertolerant zu programmieren (z.B. doppelte Speicherung von Objekten auf unterschiedlichen Knoten), ist man in der Lage, ein fehlertolerantes System aufzubauen. In einem solchen System könnten einzelne Knoten ausfallen, ohne dass der Gesamt-Cluster „abstürzt". Dieser Ansatz könnte zu einem fehlertoleranten System mit guter Performance sowie einfacher Programmierbarkeit führen.

5 Literatur

Literatur

[1] **Kai Li** *IVY: A shared virtual memory system for parallel computing* International Conference on Parallel Processing, Volume II, 94-101, **1988**

[2] **Tim Felser** *Entwurf und Implementierung von Strategien zur Prozessverwaltung in einem verteilten Linux System* Studienarbeit, Friedrich Alexander Universität, **2000**

[3] **Dominic Klumpner** *Design und Implementierung eines verteilten Dateisystems für einen teil-abschaltbaren Workstation-Cluster* Studienarbeit, Friedrich Alexander Universität, **2000**

[4] **Martin Waitz** *Design und Implementierung eines verteilten Objektmanagers für einen wartungsfreundlichen Linux Workstation Cluster* Studienarbeit, Friedrich Alexander Universität, **2000**