Skalierbare Betriebssystemkerne für Multiprozessorsysteme

Christian Brunner
Friedrich-Alexander-Universität Erlangen-Nürnberg

christian.brunner@mb.stud.uni-erlangen.de

ABSTRACT

Alle bekannten Multiprozessorbetriebssystemkerne mit gemeinsamem Speicher verwenden blockierende Synchronisation. Diese wird durch Spinlocks oder Semaphoren realisiert. Je mehr Prozessorkerne und damit gleichzeitig laufende Fäden vorhanden sind, desto eher werden diese Sperren zum Flaschenhals für das ganze System. Wenn mehrere Fäden in den gleichen kritischen Abschnitt eintreten wollen, wird die Ausführung an dieser Stelle serialisiert, was sich negativ auf den Gesamtdurchsatz auswirkt.
Zudem verursachen Sperren immer Mehraufwand. Selbst wenn kein konkurrierender Faden vorhanden ist, muss zusätzliche Arbeit erledigt werden.

Um diese Nachteile zu beseitigen, beschäftigen sich einige Forschergruppen mit skalierbaren Betriebssystemkernen für Multiprozessorsysteme. Die Ergebnisse werden im Folgenden vorgestellt.

1. EINLEITUNG

In diesem Papier werden zunächst die bekannten blockierende Synchronisationsverfahren vorgestellt. Diese sind einfach zu realisieren und sind für Einprozessorkonfigurationen auch ausreichend. Sobald diese Mechanismen jedoch auf Rechnern mit vielen Prozessorkernen eingesetzt werden, offenbart sich, dass sie für diese Umgebung nicht geschaffen wurden.
Die tatsächliche Leistung bleibt in diesem Fall weiter hinter den Erwartungen zurück, da blockierende Mechanismen nicht sehr gut mit vielen Kernen skalieren.

Ein möglicher Ausweg ist die nicht-blockierende Synchronisation. Es wird dargestellt, wie diese umgesetzt werden kann und welchen Nutzen man daraus ziehen kann.

Der Großteil dieses Papiers beschäftigt sich mit der Vorstellung zweier Betriebssystemkerne, die sich entweder durch Lockfreiheit oder Skalierbarkeit auszeichen.
1991 haben Forscher der Columbia University gezeigt, wie sie mit Synthesis einen Kernel aufgebaut haben, der komplett auf blockierende Synchronisation verzichtet. Er basiert auf „Compare-and-Swap“ Operationen.
Einen anderen Weg wählten die Entwickler von Corey. Die Lockfreiheit spielt hier eine untergeordnete Rolle. Viele wichtiger für die gute Skalierbarkeit ist die hohe Kontrolle der Applikation über die Kernaufteilung und Kernisolation.

2. ÜBERBLICK

Abschnitt 3 zeigt Probleme, die durch die Synchronisation mit Sperren auftreten können. Dabei werden die Methoden „Interrupts deaktivieren“, „Spinlocks“ und „Semaphoren“ erklärt. Zusätzlich wird auf die resultierenden Probleme „Deadlock“ und „Prioritätsumkehr“ eingegangen.
Abschnitt 4 zeigt anhand der Compare-and-Swap Operation, wie man nicht blockierend synchronisieren kann und stellt den Unterschied zwischen Wartefreiheit und nicht blockierend dar.
In Abschnitt 5 werden zwei Betriebssystemkerne vorgestellt, die sich nicht blockierender Synchronisation bedienen. Besprochen werden die Systeme „Synthesis“ und „Corey“.
Abschnitt 6 zieht ein Fazit zu skalierbaren Mulitprozessorbetriebssystemkernen.

3. PROBLEME DURCH SPERREN

Sperren sind einfach zu implementieren und sind auch relativ unproblematisch, solange man nicht mit vielen Kernen arbeitet. Wenn man aber viele Prozessorkerne hat und damit auch viele Fäden gleichzeitig laufen können, kann es zu Problemen kommen, wenn alle Fäden auf den gleichen kritischen Abschnitt im Code zugreifen wollen. Es kann dann nur jeweils ein Faden arbeiten, während alle anderen warten müssen.
Damit geht geht die zusätzliche Leistung durch mehrere Kerne zum Teil wieder verloren.

3.1 Interrupts deaktivieren

Das Deaktivieren von Hardware-Interrupts stellt die einfachste Art der Synchronisation dar. Wenn die Interrupts ausgeschalten sind, ist garantiert, dass der aktuelle Faden nicht verdrängt werden kann. Somit können die kritischen Daten, an denen gerade gearbeitet wird, nicht korrumpiert werden.
Weiterhin ist das Ein- und Ausschalten der Interrupts mit je einer Instruktion extrem billig zu realisieren.
In einem Derivat von BSD 4.3 wurden z.B. in 112 von 653 Kernel Prozeduren wichtige Datenstrukturen auf diese Weise geschützt [7].

Allerdings hat diese Möglichkeit der Synchronisation auch große Nachteile.
Zum einen dürfen Hardware-Interrupts nicht zu lange deaktiviert bleiben, damit wichtige Ereignisse wie z.B. der Clock-Interrupt nicht verloren werden. Zum anderen funktioniert das in Multicoresystemen mit gemeinsamen Speicher nicht. Deaktivierte Interrupts betreffen nur den aktuellen Kern. Hier kann der sperrende Faden tatsächlich nicht unterbrochen werden. Allerdings hat das keinen Einfluss auf die anderen Kerne. Hier können nach wie vor Fäden laufen, die die zu schützende Speicherstelle überschreiben und damit unbrauchbar machen.

3.2 Spinlocks

Spinlocks sind die einfachsten hier beschriebenen Sperren, die für Multiprozessorumgebungen geeignet sind. Jeder Faden, der in einen kritischen Abschnitt eintreten will, muss zuerst abwarten, bis die Sperrvariable freigegeben ist, und muss diese dann auf „gesperrt“ setzen. Erst dann kann die Arbeit beginnen. Abschließend wird die Sperrvariable wieder freigegeben.
Das Problem besteht hier im aktiven Warten. Der Faden probiert solange die Sperrvariable zu setzen, bis es ihm gelingt und verbraucht dadurch Prozessorleistung.
Im unwahrscheinlichen Fall, dass es mehr Prozessorkerne als Fäden gibt, würde das nichts ausmachen. In der Regel geht aber wertvolle Leistung durch aktives Warten verloren, da eigentlich lauffähige Fäden den Prozessorkern seltener zugeteilt bekommen.
Spinlocks sind ausschließlich in Mehrprozessorumgebungen interessant. Wenn nur ein Prozessorkern zur Verfügung steht, können Spinlocks nicht sinnvoll eingesetzt werden. Hier hat es keinen Sinn aktiv auf die Freigabe einer Sperre zu warten, da während des aktiven Wartens der sperrende Faden nicht arbeiten kann. Somit kann er den Spinlock auch nicht aufgeben.
In Einprozessorsystemen kommen daher statt Spinlocks z.B. Semaphoren zum Einsatz.

Listing 1 verdeutlicht die Funktionsweise von Spinlocks. Zunächst wird in einer Endlosschleife darauf gewartet, dass die Sperrvariable freigegeben wird. Wenn dieser Fall eingetreten ist, wird die Sperrvariable auf „gesperrt“ gesetzt. Damit hängen jetzt alle konkurrierenden Fäden in der Endlosschleife fest. Somit kann jetzt die kritische Arbeit beginnen. Abschließend wir die Sperrvariable freigegeben und der nächste Thread kann in den kritischen Abschnitt eintreten.

```
kritischeFunktion() {
  solange (Sperrvariable == gesperrt) {
    tue nichts;
  }
  Sperrvariable = gesperrt;
  kritische Arbeit;
  Sperrvariable = offen;
}
```

Listing 1: ”Pseudocode: Spinlock”

3.3 Semaphoren

Semaphoren beheben das Problem des aktiven Wartens der Spinlocks.
Wenn die Ressource nicht frei ist bzw. der kritische Abschnitt gesperrt ist, reiht sich der Faden in eine Liste ein und legt sich schlafen. Wenn ein anderer Faden wieder aus dem kritischen Abschnitt austritt, weckt er einen schlafenden Faden aus der Liste auf, welcher dann weiter arbeiten kann.
Damit gibt es kein aktives Warten mehr und somit wird auch nicht unnötig Prozessorlast erzeugt. Trotzdem besteht nach wie vor das Problem, dass nur ein Faden gleichzeitig den kritischen Abschnitt betreten kann.

Listing 2 skizziert den Aufbau einer Semaphore. Es gibt 2 Funktionen. Die P-Funktion reserviert eine Ressource exklusiv für einen Thread und die V-Funktion gibt diese wieder frei. Im Gegensatz zu Spinlocks kann eine Semaphore mehrere Ressourcen verwalten. Deshalb gibt es einen Zähler statt nur eine boolschen Variable.
Wenn der Zähler in der P-Funktion auf 0 steht, sind keine Ressourcen frei, und der Faden wartet passiv. Erst wenn er wieder geweckt wird, kann er den Zähler dekrementieren und mit der Ausführung fortfahren.
Die V-Funktion kann immer ausgeführt werden. Zuerst wird der Zähler wieder erhöht und anschließend wird ein Thread aufgeweckt (falls überhaupt einer wartet), damit dieser seine P-Operation beenden kann.

```
P() { // Ressource wird angefordert
  solange (Zaehler <= 0)
    warten;
  Zaehler dekrementieren;
}
V() { // Ressource wird freigegeben
  Zaehler inkrementieren;
  anderen Faden aufwecken;
}
```

Listing 2: ”Pseudocode: Semaphore”

3.4 Deadlocks und Prioritätsumkehr

Immer wenn Sperren verwendet werden, besteht grundsätzlich die Möglichkeit eines Deadlocks. Nehmen wir an, wir haben zwei Fäden (*1* und *2*) und zwei Ressourcen (**A** und **B**). Beide Fäden brauchen zwingend beide Ressourcen, um arbeiten zu können. Nehmen wir weiter an, Faden *1* allokiert zuerst Ressource **A** und wird dann von Faden *2* verdrängt. Dieser allokiert allerdings zuerst Ressource **B** und kann dann nicht weitermachen, da **A** geblockt ist. Auch Faden *1* kann nicht arbeiten, da ihm **B** fehlt.
Die Lösung ist offensichtlich: Wenn die Ressourcen immer in eine festgelegten Reihenfolge allokiert werden, kann dieser Fall nicht eintreten. Je komplexer das System wird, desto schwieriger ist es, dieses zu kontrollieren.

Ein weiteres Problem stellt die unbeschränkte Prioritätsumkehr dar. Nehmen wir an, ein niedrig priorer Faden hält eine Ressource und wird von einem hoch prioren Faden verdrängt, der die gleiche Ressource benötigt. In diesem Fall kann der Faden mit der höheren Priorität nicht arbeiten und muss auf den Faden mit niedrigerer Priorität warten.
Auch für dieses Problem gibt es Lösungen. Es zeigt jedoch,

dass man auf sehr viel achten muss, wenn man mit Sperren arbeitet.

4. NICHT BLOCKIERENDE SYNCHRONISATION

Die oben beschriebenen Probleme lassen sich durch blockierungsfreie Synchronisation verhindern. Diese Art der Synchronisation basiert z.B. auf „Compare-and-Swap"-Operationen (CAS), die in einem atomaren Arbeitsschritt einen Variableninhalt mit einem Wert vergleichen und im Erfolgsfall diesen auch ändern können.
Es existieren neben CAS auch noch andere technische Lösungen, die im Folgenden aber nicht berücksichtigt werden.

Neben dem einfachen Einwort-Compare-and-Swap gibt es auch Mehrwort-Varianten. Mit deren Hilfe kann man auf mehrere Speicheradressen atomar zugreifen. Zum Beispiel ist es möglich bei doppelt verketteten Listen beide Zeiger auf ein Listenelement gleichzeitig zu manipulieren.
Allgemein lassen sich mit Mehrwort-Varianten grundlegende Datenstrukturen wie Stapel (**Stacks**), Warteschlangen (**Queues**) und verkettete Listen (**Linked Lists**) erstellen, die dann komplett ohne blockierende Verfahren manipuliert werden können.
Diese Varianten von CAS sind allerdings nicht flächendeckend verfügbar, so dass man diese Funktionalität in der Regel nicht vorraussetzen darf.
Bei Synthesis kommt CAS2 zum Einsatz, was ein Zweiwort-CAS-Variante ist.

In Listing 3 kann man erkennen, wie die CAS-Operation funktioniert.
Es wird überprüft, ob der Wert `compare` mit dem Speicherinhalt von `mem_addr` übereinstimmt. Wenn das der Fall ist, wird der Speicheradresse `mem_addr` der Wert `update` zugewiesen und es wird signalisiert, dass die Operation geglückt ist. Wenn der Vergleich jedoch fehlschlägt wird Misserfolg signalisiert.

```
bool CAS (compare, update, mem_addr) {
  if (*mem_addr == compare) {
    *mem_addr = update;
    return SUCCESS;
  }
  return FAIL;
}
```

Listing 3: "CAS (1 Wort)"

Definition: Lockfreiheit

Laut Herlihy gibt es die Abstufung wartefrei (wait-free) und nicht blockierend (non-blocking)[1]. Die Anforderung wartefrei ist strenger und verlangt, dass *jeder* Faden in einer endlichen Zeit fertig wird. Nicht blockierend bedeutet lediglich, dass es *einen* Faden gibt, der in endlicher Zeit fertig wird.[4]
Der Unterschied ist also, dass Wartefreiheit Verhungern ausschließt.
Das Verhungern in Betriebssystemkernen ist jedoch sehr unwahrscheinlich und Wartefreiheit sehr teuer, deshalb wird in den vorgestellten Systeme nicht blockierende Synchronisation verwendet.
Zudem ist Wartefreiheit hauptsächlich für Echtzeitsysteme wichtig. In nicht-echzeitfähigen Systemen ist die Gefahr der Verhungerung tolerierbar.

[1]Lockfrei (lock-free) wird hier synonym zu nicht blockierend verwendet.

5. BETRIEBSSYSTEMKERNE

Im Folgenden werden zwei Betriebssystemkerne vorgestellt, die besonders gut mit Mehrkernprozessoren skalieren. Dazu kommen nicht blockierende Synchronisation und feingranulare Aufteilung der Ressourcen zum Einsatz.
Die „Lockfreiheit" betrifft jeweils nur den Betriebssystemkern. Für Applikationen werden durchaus auch blockierende Synchronisationsmethoden bereitgestellt, deren Implementierung jedoch nicht blockierende ist.
Synthesis V.1 wurde an der Columbia University entwickelt. Die Veröffentlichung stammt von 1991 [7]. Damit ist Synthesis das älteste hier erläuterte System.
Corey ist dagegen ein sehr neuer Ansatz aus dem Jahr 2008, der hauptsächlich am MIT und der Fudan University entstanden ist [1].

5.1 Synthesis

Das Ziel bei der Konzeption und Entwicklung von Synthesis war es, ausschließlich auf nicht blockierende Synchronisation zu setzen. Zu keinem Zeitpunkt kommen Spinlocks oder Semaphoren zum Einsatz. Außerdem sollte die Implementierung möglichst performant sein.
Damit schied der generelle Ansatz von Herlihy, mit dem man alle Arten von sequenziellen Implementationen von Datenstrukturen mit Hilfe von CAS in wartefreie umwandeln kann, aus. Um einen Stapel zu verändern, wird nach Herlihy der ganze Stapel kopiert, verändert und anschließend wird der Zeiger auf den alten Stapel mit Hilfe von CAS auf den neuen Stapel umgeschrieben.[4]
Dieser Ansatz stand dem Ziel eines Kernels mit möglichst geringen Overhead im Weg.

Der erste Ansatz war, möglichst alle gemeinsam genutzten Daten in ein oder zwei Wörter zu packen. Diese kann man dann direkt mit CAS und CAS2 manipulieren. Für größere Daten werden spezielle LIFO Stapel, FIFO Warteschlangen und verkettete Listen erstellt.
Wenn die Daten auch in diese Objekte nicht passen, bedient man sich anderer Methoden: Am Anfang eines kritischen Abschnitts wird von jedem Faden der Systemstatus in einem Wort kodiert. Das passiert so, dass jeder Faden zu einem anderen Ergebnis kommt. Am Ende des kritischen Abschnitts wird kontrolliert, ob sich der Systemstatus geändert hat. Wenn das nicht der Fall ist, hat kein anderer Faden gestört.
Generell wurde viel Arbeit investiert, um kritische Abschnitte möglichst kurz zu halten. Außerdem hat es sich als sinnvoll erwiesen, kritische Abschnitte in mehrere kürzere aufzuteilen.

Der Synthesiskern ist aus sogenannten **Quajects** aufgebaut. Quajects sind Codefragmente mit Datenstrukturen.
Sie können z.B. Instanzen von abstrakten Datentypen wie Stapeln, Warteschlangen oder verketteten Listen sein. Alternativ können sie auch Betriebssystemabstraktionen wie Fäden, Speichersegmente oder E/A-Geräte reprästentieren.

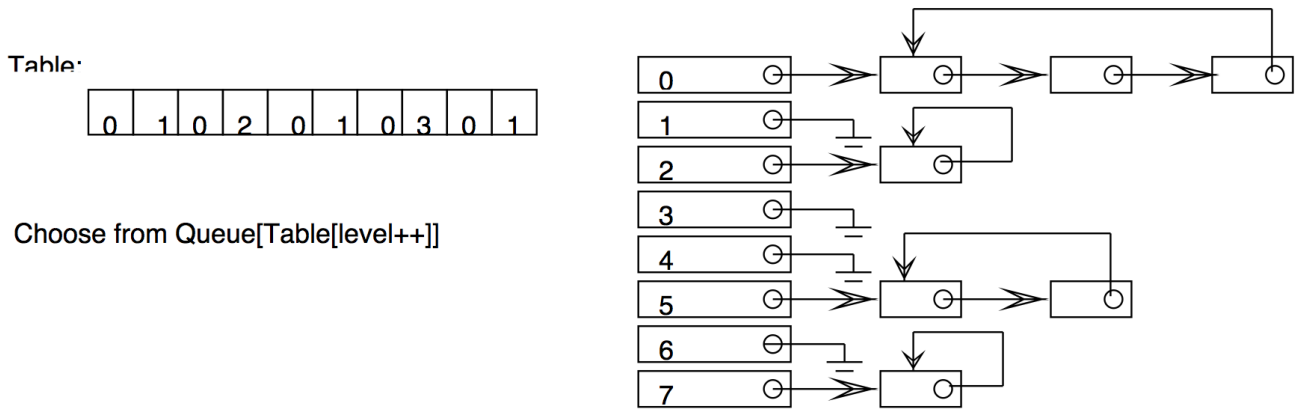

Figure 1: Fadenverwaltung in Synthesis

Listing 4 zeigt, wie ein Stapel in Synthesis aufgebaut ist. Wie man sieht ist die Push-Funktion zwar aufwendiger, trotzdem lässt sich auch diese mit einem CAS2 realisieren.
FIFO Warteschlangen sind komplexer als Stapel. Neben der Get- und Put-Funktion gibt es Rückruffunktionen für den Fall, dass die Warteschlange voll oder leer ist.
Des Weiteren gibt es die jeweils optimale Implementierung einer Warteschlange für die verschiedenen Konstellationen aus einem oder mehreren Produzenten und einem oder mehreren Konsumenten. Dadurch kommt es bei der Konstellation n-Produzenten und 1 Konsument zu einer Pfadlänge der Put-Funktion von nur 11 Instruktionen auf einem MC68030 Prozessor.

In der Push-Funktion wird zunächst der alte Stapelzeiger gespeichert (`old_SP`). Der neue Stapelzeiger (`new_SP`) zeigt auf die Speicherstelle direkt hinter `old_SP`. Danach wird der Speicherinhalt vom neuen Stapelzeiger in `old_val` gespeichert.
Bei der CAS2-Operation wird überprüft, ob `old_SP` identisch mit dem Speicherinhalt von `SP` ist. Zeitgleich wird überprüft, ob der Wert `old_val` mit dem Speicherinhalt von `new_SP` übereinstimmt.
Wenn beide Tests positiv verlaufen sind, wird `old_SP` druch `new_SP` und `old_val` durch `elem` ersetzt.
Wenn mindestens ein Test negativ verlaufen ist, wurde die Funktion unterbrochen und es muss von vorne angefangen werden.

Die Pop-Funktion ist nicht so komplex, da hier ein einfacher CAS-Befehlt ausreicht. Auch hier wird zunächst der alte Stapelzeiger in (`old_SP`) gespeichert. Der neue Stapelzeiger (`new_SP`) zeigt auf die Speicherstelle direkt vor `old_SP`. Die Rückgabevariable bekommt den Speicherinhalt von (`old_SP` zugewiesen.
In der CAS-Operation wird überprüft, ob `old_SP` noch mit `SP` übereinstimmt. Wenn das der Fall ist, ist die Funktion nicht unterbrochen wurden und der alte Stapelzeiger wird durch den neuen ersetzt. Andernfalls muss von vorne angefangen werden.

```
void Push (elem) {
  retry:
    old_SP = SP;
    new_SP = old_SP - 1;
    old_val = *new_SP;
    if (CAS2(old_SP,old_val,new_SP,
             elem,&SP,new_SP) == FAIL)
      goto retry;
}
elem Pop (elem) {
  retry:
    old_SP = SP;
    new_SP = old_SP+1;
    elem = *old_SP;
    if (CAS(old_SP,new_SP,&SP) == FAIL)
      goto retry;
    return elem;
}
```

Listing 4: "Quaject: Stapel"

Auch das Dateisystem ist aus einer Reihe von ineinandergreifenden Quajects realisiert. Ganz unten sitzt der Treiber für die Festplatte. Dieser schickt die Daten an den Dateimapper. Dessen Aufgabe ist es, die Adressen in Zylinder und Sektoren der Festplatte umzurechnen und umgekehrt. Der Dateimapper reicht dann die Daten an den Dateipuffer weiter. Dieser implementiert die UNIX-artigen Systemaufrufe `read`, `write` und `seek`. Ganz oben sitzt das Dateisystem Quaject.
Die Kommunikation zwischen Treiber und Dateimapper ist synchron und funktioniert über Koroutinen. Die Kommunikation zwischen Dateipuffer und Dateimapper ist asynchron mit nicht blockierenden Warteschlangen realisiert.

Jeder Faden in Synthesis wird durch einen **Thread Table Entry** (TTE) beschrieben. In diesem TTE sind die Ein- und Auslagerungsroutinen sowie der Fadenkontext hinterlegt.
Beim Scheduling gibt es feste Prioritäten. Jede Priorität hat ihre eigene Warteschlange. Die höchste Priorität (Level 0) erhält 50% der Prozessorzeit. Priorität 1 erhält die Hälfte der übrigen Zeit usw.
In *Abbildung 1* sieht man auf rechten Seite die Warteschlan-

gen der einzelnen Prioritäten. In diesem Beispiel sind nur Threads für die Prioritäten 0, 2, 5 und 7 dargestellt.
Auf der linken Seite sieht man, in welcher Reihenfolge die Threads ausgewählt werden. Dazu wird eine statische Tabelle verwendet, in der gespeichert ist, wann welche Warteliste bedient wird. Diese spiegelt die oben erklärte 50%-Regel wieder.
In diesem Beispiel kommen die Prioritäten also ich der Reihenfolge 0, 1, 0, 2, 0, ... etc. zum Zug.

Fazit

Synthesis zeigt, dass es durchaus machbar ist, bei der Implementierung eines Betriebssystemkerns komplett auf blockierende Mechanismen zu verzichten.
Durch die Verwendung der CAS2-Operation hat man jedoch in Kauf genommen, dass Synthesis nur auf eine kleinen Auswahl an Prozessoren funktionsfähig ist.
Die Leistung von Synthesis ist sehr schwer einzuschätzen, da in den referenzierten Papieren keine vergleichenden Benchmarks verfügbar waren.

5.2 Corey

Corey orientiert sich am Prinzip des Exokernels.
Ein Exokernel beschränkt sich auf möglichst wenige Abstraktionen. Somit sind z.B. direkte Zugriffe auf Speicherblöcke, Festplattenbereiche, etc. möglich. Die Aufgabe des Exokernels besteht darin, die Hardware vor konkurrierenden Zugriffen zu schützen.[2]
Weitergehende Funktionen werden dann von einem Bibliotheksbetriebssystem bereitgestellt.

5.2.1 Systembeschreibung

Corey stellt die low-level Abstraktionen „Freigaben“ (**shares**), „Adressbäume“ (**address trees**) und „Kernelkerne“ (**kernel cores**) bereit, die von Bibliotheksbetriebssystemen und Applikationen genutzt werden können.
Freigaben erlauben den Applikationen präzise Kontrolle darüber, welche Kernelressourcen mit anderen Prozessorkernen geteilt werden.
Adressbäume bieten Programmen die Möglichkeit zu entscheiden, welche Seitentablleneinträge nur für einen Kern sichtbar sind, und welche global verfügbar sein sollen.
Kernelkerne geben den Applikationen die Möglichkeit, den Kernel auf einem dedizierten physikalischen Prozessorkern laufen zu lassen. Dadurch können Systemaufrufe statt über Traps mit shared-memory IPCs realisiert werden, was einen erheblichen Geschwindigkeitsgewinn verspricht.

Im Gegensatz zu Synthesis steht bei Corey nicht so sehr die Lockfreiheit im Vordergrund. Viel interessanter ist hier die Konfigurierbarkeit durch die jeweilige Applikation.
So können diese Applikationen festlegen, welche Daten mit anderen Kernen geteilt werden und wie die zu erledigende Arbeit auf die verfügbaren Kerne aufgeteilt wird.

Freigaben. Eine wichtige Eigenschaft der Coreykerns ist, dass keine Kerneldatenstrukturen mit anderen Prozessorkernen geteilt werden, sondern dem aktuellen Kern lokal zugeordnet sind. Nur Applikationen können Datenstrukturen gezielt freigeben. Damit kommt es von vornherein zu weniger Konkurrenzsituationen, da es weniger Datenstrukturen gibt, die von mehreren Kernen parallel benutzt werden.
Das ist ein erheblicher Vorteil gegenüber herkömmlichen Betriebssystemen. Untersuchungen haben gezeigt, dass in derartigen Systemen bereits auf einem Zweikernprozessor die Konkurrenzsituation um die Warteschlange der lauffähigen Fäden einen beachtlichen Anteil zur Laufzeit der Programme beitragen kann.[3]

Kernverwaltung. Corey kann auch Prozessorkerne für einen langen Zeitraum komplett einzelnen Applikationen überlassen, welche die Kerne dann selbst verwalten.
Das bietet zwei Vorteile: Zum einen fallen Kontextwechsel dann weg und zum anderen muss sich der Kernel nicht mehr um das Scheduling kümmern. Das Bibliotheksbetriebssystem kann dann eine eigene, feinere Schedulingstrategie verwirklichen.
Besonders interessant wird diese Funktionalität dann, wenn die Prozessorentwicklung soweit fortgeschritten ist, dass auf einem Prozessor mehr Kerne als Prozesse im Betriebssystem vorhanden sind.

Speicherverwaltung. Der Kernel bietet den Speicher in Form von Segmenten an. Wenn das Bibliotheksbetriebssystem Segmente anfordert, sind diese zunächst nur für den anfordernden Kern sichtbar. Bei Bedarf können die Segmente jedoch anderen Kernen zur Verfügung gestellt werden.
Um seinen Adressraum zu definieren, benutzt ein Bibliotheksbetriebssystem Adressbäume.

Geräte. Geräte werden dem Bibliotheksbetriebssystem als eine Liste von Geräteobjekte präsentiert. Sobald ein Gerät angefordert wird, wird exklusiver Zugriff darauf sichergestellt.
Die Kommunikation mit dem Geräte funktioniert über eine Segmentfreigabe.

Bibliotheksbetriebssystem. Das Bibliotheksbetriebssystem stellt der Applikation die Funktionalität zur Verfügung, die der Kernel nicht bietet.
Dem Prinzip des Exokernels folgend bietet der Coreykernel zwar Zugriff auf die Netzwerkhardware, er bietet jedoch keinen Protokollstapel an. Diese Aufgabe übernimmt das Bibliotheksbetriebssystem. Wenn die Anzahl der Protokallstapel die Anzahl der physikalischen Netzwerkgeräte übersteigt, kann der Coreykern weitere Netzwerkgeräte virtualisieren.
Die Empfangs- und Senderingpuffer der Netzwerkkarte werden in diesem Fall geteilt. Geregelt wird der Zugriff dann mit Spinlocks.
Das Bibliotheksbetriebssystem stellt auch die Funktion `cfork` bereit, die sich an der UNIX Funktion `fork` orientiert. Mit Hilfe dieser Funktion kann ein Kindprozess auf einem anderen Prozessorkern ausgeführt werden. Im Normalfall teilen Kind- und Elternprozess wenig Daten. Der aufrufende Prozess markiert die meisten Segmente seines Wurzeladressbaums als „copy-on-write“ und blendet diesen im Wurzeladressbaum des Kindprozesses ein.

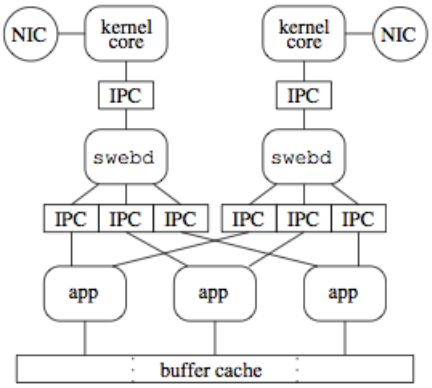

Figure 2: Webserver Konfiguration

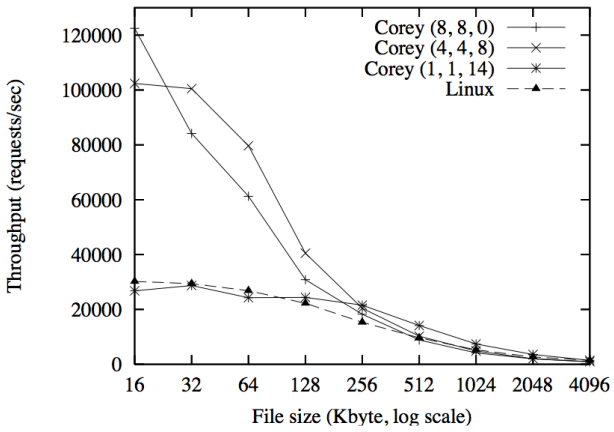

Figure 3: Durchsatz filesum. Die Konfiguration „Corey (8,8,0)“ bedeutet z.B. 8 Swebd Kerne, 8 Kernelkerne und 0 Filesumkerne (in diesem Fall wird filesum vom Swebd abgearbeitet)

5.2.2 *Benchmarks*

Um die Performance von Corey zu testen, haben sich die Entwickler für verschiedene Szenarien entschieden, auf deren Basis Benchmarks erstellt worden sind.

Webserver. Der Webserver ist wie folgt aufgebaut (siehe auch *Abbildung 2*):
Kernel Core: Die beiden Kernel Cores wurden von den zugehörigen Webservern (**Swebd**) gestartet und verwalten je eine Netzwerkschnittstelle und stellen Systemaufrufe über IPC zur Verfügung.
Swebd: Jeder Prozessorkern eines **Swebd** hat einen eigenen Netzwerkstapel. Der **Swebd** behandelt alle HTTP-Anfragen und reicht die Anfragen an die Applikation weiter.
App: Die Applikation arbeitet die Anfrage ab, und schickt das Ergebnis wieder an den **Swebd** zurück. Dabei wird darauf geachtet, dass der Kern die Anfrage bearbeitet, welcher mit der höchsten Wahrscheinlichkeit die nötigen Daten schon im lokalen Cache vorliegen hat. Falls die nötigen Daten nicht lokal vorliegen, können sie über den Puffercache geholt werden, was aber mit Performancenachteilen belegt ist.

Jedem Kernel Core, **Swebd** und **App** steht ein Prozessorkern zur Verfügung. Da die einzelnen Kerne unabhängig sind (jeder **Swebd** hat seinen eigenen Netzwerkstapel, Packetpuffer, etc.) und die einzige Schnittstelle zwischen den Kernen die IPCs sind, skaliert diese Lösung sehr gut.

Um den Webserver mit einem Linuxserver zu vergleichen wurde unter anderem ein **„filesum“**- Benchmark ausgeführt: Die angeforderte Datei wird geöffnet und der Inhalt zurückgeschickt.
Es wurden drei verschieden Corey-Konfigurationen mit einer Linuxkonfiguration verglichen. Bei kleinen Dateien ist es offensichtlich effektiver, wenn **Swebd** die Aufgabe von **App** übernimmt, und die Dateien auch gleich ausliest.
Je größer die Dateien allerdings werden, desto mehr Sinn macht es diesen Vorgang auf dedizierte Kerne auszulagern. Gerade bei kleinen Dateien ist Corey in der richtigen Konfiguration Linux deutlich überlegen (siehe *Abbildung 3*).

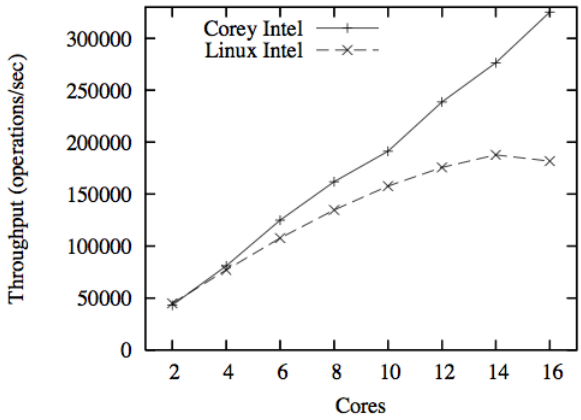

Figure 4: TCP Durchsatz

TCP Benchmark. Ein TCP-Benchmark (*Abbildung 4*) zeigt, dass Corey fast perfekt mit der Anzahl der Prozessorkerne skaliert. Bei Linux hingegen steigt der Durchsatz langsamer und fängt bei 16 Kernen sogar wieder an zu fallen. Das liegt an zwei globalen Spinlocks im Linuxkernel beim Erstellen und Freigeben eines Sockets. Diese beiden Sperren (`inode_lock` und `dcache_lock`) werden selbst dann benutzt, wenn ein Socket ausschließlich von einem Prozessorkern verwendet wird. Das Problem liegt hier also in der zentralen Verwaltung aller Ressourcen.
Bei Corey gibt es diese Probleme nicht, weil sich jeder Kernelkern um seinen eigenen Ethernetadapter kümmert. Es wird also statt einer globalen Ressourcenverwaltung eine lokale verwendet. Globale Sperren gibt es hier demnach nicht.

MapReduce. Der dritte Benchmark mit dem die Performance von Corey getestet wurde, beruht auf MapReduce. MapReduce ist ein Framework mit dem parallele Berechnungen über große Datenmengen vollzogen werden können. Der

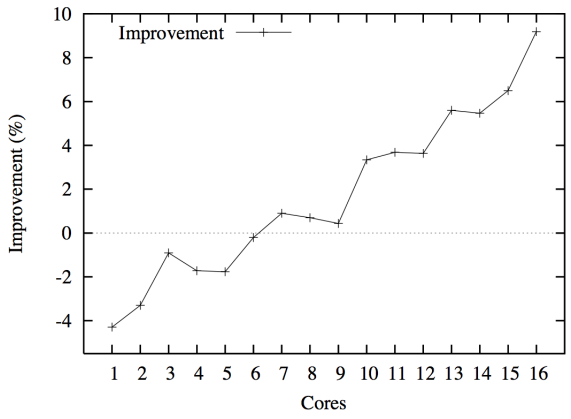

Figure 5: wc Leistung von Corey gegenüber Linux

Programmierer braucht dazu keine Expertenkenntnisse um gute Resultate zu erreichen.
Als Ausgangspunkt diente die Phoenix Implementation des Algorithmus. Diese wurde an Corey angepasst und stark verbessert. Bei einer Konfiguration mit 16 Kernen konnte die Leistung um den Faktor vier verbessert werden.
Für einen konkreten Test wurde das Programm `wc` (wordcount) verwendet, das die Anzahl der Wörter in einer 1 GB großen Datei feststellt.
Abbildung 5 zeigt, wie sich die Leistung relativ zu Linux verhält. Ab ca. sechs Kernen ist Corey schneller und kann den Vorsprung bei 16 Kernen auf ca. 9,2% ausbauen.
Der Grund für den Vorsprung mit 16 Kernen ist, dass Linux 11% der map-Phase und 17% der reduce-Phase in der Funktion `smp_invalidate_interrupt` verbringt. Diese Funktion leert TLB-Einträge, wenn Speicher aus dem gemeinsamen Adressraum freigegeben wird. Corey verhindert unnötiges Invalidieren der TLBs, weil jeder Prozessorkern den Stapel des aktuell laufenden Fäden in einen kernlokalen Adressraum schiebt.

5.2.3 Fazit

Zur Leistungssteigerung durch mehrere Prozessorkeren verlässt sich Corey nicht auf einzelne technische Aspekte. Vielmehr resultiert die Skalierbarkeit aus eine Kombination aus nicht blockierender Synchronisation und hoher Kontrolle der Applikation über die Hardware.
Im Gegensatz zu Synthesis wird nicht ausschließlich nicht blockierende Synchronisation verwendet.
Vor allem wenn sehr viele Prozessorkerne (mehr als 16) zur Verfügung stehen, kann Corey zeigen, dass es konventionellen Betriebsystemen wie Linux teilweise überlegen ist.
Der TCP-Benchmark bei Corey hat gezeigt, dass bei Linux mit 14 Kernen das Leistungsmaximum erreicht ist, bei mehr Kernen geht die Gesamtleistung sogar zurück.

6. FAZIT

Aktuell geht der Trend in der Prozessorentwicklung dahin, dass immer mehr Kerne auf einem Prozessor implementiert werden, diese dafür aber höchstens gleich schnell oder sogar langsamer als Einkernprozessoren getaktet werden.
Von einem steigenden Takt kann fast jede Applikation profitieren. Bei mehreren Kernen sieht das anderes aus. Die Applikationen und das Betriebssystem müssen parallel programmiert sein. Aber auch das reicht noch nicht aus. Es muss auch sichergestellt sein, dass das Betriebssystem die vielen Kerne effizient auslasten kann. Die Skalierung muss also möglichst gut sein. Es ist nicht zielführend, wenn ein Prozessor beispielsweise 100 Kerne hat, das Betriebssystem aber nur bis zu 10 Kerne skaliert.

Die hier vorgestellten Konzepte beheben auf ihre Art und Weise diesen Mangel.
Synthesis setzt dazu auf ausschließlich nicht blockierende Synchronisation. Corey verwendet diese Technik zwar auch, lässt aber auch Spinlocks zu. Dazu kommt bei Corey noch die hohe Kontrolle der Applikation über die Hardware.
Jedoch sind diese Systeme bis jetzt nur Forschungskonzepte und es ist unklar ob diese jemals auf produktiv arbeitenden Rechnern und Servern eingesetzt werden.
Fakt ist jedoch, dass es absolut notwendig ist, an neuen, skalierbaren Betriebssystemkonzepten zu forschen. Andernfalls wird die Software in Zukunft nicht mehr von der immer größeren Vielzahl an Prozessorkernen profitieren können. Da die Prozessorentwicklung mit den Taktraten aber schon an die Grenze des physikalsich Machbaren stößt, führt kein Weg an der Parallelisierung vorbei.

7. REFERENCES

[1] S. Boyed-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. December 2008.

[2] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[3] C. Gough, S. Siddha, and K. Chen. Kernel scalability - expanding the horizon beyond fine grain locks. *Proceedings of the Linux Symposium 2007*, June 2007.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, January 1991.

[5] M. Hohmuth and H. Härting. Pragmatic nonblocking synchronization for real-time systems. 2001.

[6] H. Massalin. Synthesis:an efficient implementation of fundamental opteratin systems. 1992.

[7] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. *Technical Report No. CUCS-005-91*, June 1991.

[8] C. Pu and H. Massalin. Quaject composition in the sysnthesis kernel. 1991.