Eine Maschinenunabhängige Portierung des MPD- und SR-Laufzeitsystems auf NetBSD

Ignatios Souvatzis

Institut für Informatik, Abt. V

universitätbonn

<ignatios@cs.uni-bonn.de>

Einführung: Was sind MPD und SR?

universitätbonn

Was sind MPD und SR?

- MPD ist eine Sprache für „Multithreaded, Parallel and Distributed programming“ [3]
- ähnlich zu C
- ergänzt um Sprachelemente für
  - multitasking
  - Prozess-Synchronisation
  - Prozesskommunikation
- SR (“Synchronizing Resources”) ist der Vorgänger - eher PASCAL-ähnlich - zuerst 1988[2, 1] beschrieben
- Compiler:
  - implementiert als Präprozessor, der C erzeugt

universitätbonn

Das Laufzeitsystem von MPD/SR

- Das Laufzeitsystem übersetzt Prozesse der Sprache entweder auf
  - Threads, die von (meist) Assemblerroutinen innerhalb eines Unix- Prozesses auf einer Einzelprozessormaschine erzeugt werden
  - Threads, die von maschinenabhängigem Code auf einer Multiprozessormaschine mit gemeinsamem Speicher gebildet werden
  - wie oben, aber über ein (lokales) Netzwerk verteilt

Portierungsprobleme (entf. in SR-2.3.3 /MPD-1.0.1)

Verifizierung und Leistungsmessung

universitätbonn

Verifizierungsmethoden

MPD enthält zwei Verifizierungswerkzeuge:

- *verification suite*
  - ein frischübersetztes oder installiertes MPD prüfen
    `mpdv/mpdv [-p] [-v]`
  - bei SR ist vsuite aufgeteilt
- Einzeltest für die Kontextwechselprimitive
  - zum Verifizieren neuer Portierungen
  - Vom MPD-Makefile automatisch ausgeführt
  - oder:

```
cd csw
make cstest
./cstest
```

Leistungsmessung

MPD enthält zwei Werkzeuge zur Leistungsmessung:

- Einzelmessung der Kontextwechselprimitive
  `cd csw`
  `make csloop`
  `./csloop N` (N ist die gewünschte Laufzeit in Sekunden)
- für das Gesamtsystem in vsuite:
  `cd vsuite/timings`
  `sh build.sh`
  `sh run.sh`
  `sh report.sh`

Testmaschinen

Testmaschine	A	B
Architektur	i386	arm
CPU	Pentium 4	SA-110
clock	3200 MHz	233 MHz
cache	2 MB	16kB I + 16 kB D

Implementierungsstrategie

universitätbonn

Vom SR-Port kopieren? [6]

- MPD ist im wesentlichen eine andere Syntax für die Sprachelemente von SR.
- Internes der Implementierung ist “essentially the same” (README von MPD)
- vielleicht können Codepatches übernommen werden?
  - viele Patches konnten nach `s/sr/mpd/g` übernommen werden.
- Das Laufzeitsystem hätte von den Autoren ausgeklammert werden sollen.
  - zuviel Arbeit für mich - zumal bei Änderungen im Ursprungspaket.

universitätbonn

Vergleichsimplementierung

Zwei Implementierungen waren einfach zu machen, und wurden zum Testen und als Vergleichsbasis benutzt:

- Eine i386-spezifische Implementierung, die mitgelieferte Assemblerroutinen benutzt
  (Funktioniert gut - alle Tests waren erfolgreich.)
- Eine maschinenunabhängige Implementierung, die eine leicht abgeänderte Version des mitgelieferten `svr4.c` benutzt.
  Verwendet werden die von SVR4 abgeleiteten Systemaufrufe *getcontext()* und *setcontext()*.

universitätbonn

Reine Kontextwechselzeiten

Implementierung	A	B
Assembler	0.013 μs	n/a
SVR4 - Systemaufrufe	1.453 μs	9.649 μs

Systemleistung, system A

Testbeschreibung	i386 ASM	SVR4 s.c.
loop control overhead	0.002 μs	0.002 μs
local call, optimised	0.011 μs	0.011 μs
interresource call, no new process	0.270 μs	0.250 μs
interresource call, new process	0.650 μs	4.350 μs
process create/destroy	0.540 μs	4.280 μs
semaphore P only	0.011 μs	0.011 μs
semaphore V only	0.008 μs	0.008 μs
semaphore pair	0.019 μs	0.019 μs
semaphore requiring context switch	0.110 μs	1.550 μs
asynchronous send/receive	0.300 μs	0.300 μs
message passing requiring context switch	0.400 μs	1.920 μs
rendezvous	0.600 μs	4.200 μs

Systemleistung, system B

Testbeschreibung	ARM ASM	SVR4 s.c.
loop control overhead	n/a	0.056 μs
local call, optimised	n/a	0.355 μs
interresource call, no new process	n/a	4.080 μs
interresource call, new process	n/a	55.900 μs
process create/destroy	n/a	58.780 μs
semaphore P only	n/a	0.301 μs
semaphore V only	n/a	0.249 μs
semaphore pair	n/a	0.487 μs
semaphore requiring context switch	n/a	11.180 μs
asynchronous send/receive	n/a	5.190 μs
message passing requiring context switch	n/a	30.140 μs
rendezvous	n/a	54.000 μs

universitätbonn

Zwischenbetrachtung

Auf dem benutzten Ethernet beträgt die Zweiwegverzögerung etwa $200\mu s$.

- D.h., die SVR4-artige Implementierung ist gut genug, ausser für noch langsamere Maschinen.

Eine weitere Implementierungsidee

universitätbonn

Kontextwechselprimitive aus libpthread

libpthread enthält *_getcontext_u()*, *_setcontext_u()* und *_swapcontext_u()*

- wie getcontext/setcontext/makecontext, aber ohne Kernelaufruf
- libpthread selbst verläßt sich auf „Scheduler Activations“ für den Fall, dass Kernelhilfe erforderlich ist[5]
- Benutzen dieser Routinen sollte uns die Leistung der Assemblerimplementierung liefern, ohne dass wir selbst Assemblercode schreiben müssen.

universitätbonn

Kontextwechselprimitive aus libpthread

Drei Probleme:

- i386 benutzt Funktionszeiger, und die oben genannten Namen sind Makros in einer nichtöffentlichen Includedatei.
  - öffentlich definieren
- Wir müssen die Objektmodule aus `libpthread.a` entnehmen, da libpthread einige schreckliche Dinge mit Programmen anstellt, die es nicht initialisieren.
  - z.B. `ar x`, oder `netbsd.o` gegen `libpthread.a` linken und das Ergebnis weiterverwenden
- wir brauchen immer noch einen Systemaufruf für die Erzeugung eines Kontextes

universitätbonn

Reine Kontextwechselzeiten

Implementierung	A	B
Assembler	0.013 μs	n/a
. . . context_u libary calls	0.138 μs	0.237 μs
SVR4 system calls	1.453 μs	9.649 μs

universitätbonn

Systemleistung, System A

Testbeschreibung	i386 ASM	context_u	SVR4 s.c.
loop control overhead	0.002 μs	0.002 μs	0.002 μs
local call, optimised	0.011 μs	0.011 μs	0.011 μs
interresource call, no new process	0.270 μs	0.260 μs	0.250 μs
interresource call, new process	0.650 μs	4.200 μs	4.350 μs
process create/destroy	0.540 μs	4.020 μs	4.280 μs
semaphore P only	0.011 μs	0.011 μs	0.011 μs
semaphore V only	0.008 μs	0.008 μs	0.008 μs
semaphore pair	0.019 μs	0.019 μs	0.019 μs
sem. req. context switch	0.110 μs	0.220 μs	1.550 μs
asynchronous send/receive	0.300 μs	0.290 μs	0.300 μs
mess. pass. req. context switch	0.400 μs	0.560 μs	1.920 μs
rendezvous	0.600 μs	0.850 μs	4.200 μs

universitätbonn

Systemleistung, System B

Testbeschreibung	ARM ASM	context_u	SVR4 s.c.
loop control overhead	n/a	0.057 μs	0.056 μs
local call, optimised	n/a	0.376 μs	0.355 μs
interresource call, no new process	n/a	4.300 μs	4.080 μs
interresource call, new process	n/a	27.250 μs	55.900 μs
process create/destroy	n/a	25.240 μs	58.780 μs
semaphore P only	n/a	0.304 μs	0.301 μs
semaphore V only	n/a	0.254 μs	0.249 μs
semaphore pair	n/a	0.506 μs	0.487 μs
sem. req. context switch	n/a	1.570 μs	11.180 μs
asynchronous send/receive	n/a	5.550 μs	5.190 μs
mess. pass. req. cont. switch	n/a	6.740 μs	30.140 μs
rendezvous	n/a	9.600 μs	54.000 μs

universitätbonn

Diskussion: Leistung auf A

Auf der 3200-MHz-Maschine beobachten wir drei Arten von Operationen:

1. Operationen ohne Kontextwechsel sind für alle Implementierungen gleich schnell.
2. Operationen, die einen neuen MPD-Prozess erzeugen, sind mit der neuen Implementierung etwa so schnell wie mit der SVR4-artigen Implementierung.
3. Andere Operationen sind langsamer als im Assemblerfall, aber schneller als für SVR4-Aufrufe, um die Zeit von etwa ein oder zwei Kontextwechselzeiten

universitätbonn

Diskussion: Leistung auf B

- Auf der 233-MHz-Maschine können wir etwa die gleichen Beobachtungen machen wie auf der schnelleren Maschine.
- SVR4-artige Prozesserzeugung/vernichtung/synchronisation benötigt etwa $\frac{1}{3}$ der doppelten Netzwerkverzögerung
  - für Maschinen mit einem kleineren Takt als etwa 300 MHz, sollte unsere neue Implementierung (oder eine Assemblerbasierte) verwendet werden.

Schlußfolgerungen

- lokaler Fall, sehr schnelle Maschine: ...context_u ist akzeptabel, Assembler ist etwas schneller.
- vernetzter Fall, schnelle Maschine: ...context_u ist schnell genug, SVR4 wäre so gerade akzeptabel.
- vernetzter Fall, langsame Maschine: ...context_u oder Assembler notwendig, SVR4 unakzeptabel.

universitätbonn

Ausblick

- eine Implementierung von MPDMULTI
  - sollte nicht zu schwierig zu schreiben und verifizieren sein, nachdem inzwischen zumindest HT-Maschinen bei uns verfügbar sind.
- weitere Betriebssysteme (MacOS-X? Solaris?)

Literatur

Literatur

[1] Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, 1993

[2] Gregory R. Andrews, Ronald A. Olsson, Michael H. Coffin, Irving Elshoff, Kelvin D. Nilsen, Titus Purdin and Gregg M. Townsend, *An Overview of the SR Language and Implementation*, 1988, ACM TOPLAS Vol. 10.1, p. 51-86

[3] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming,* Addison-Wesley, 2000 (ISBN 0-201-35752-6)

universitätbonn

Literatur (2)

[4] Gregg Townsend, Dave Bakken, *Porting the SR Programming Language*, 1994, Department of Computer Science, The University of Arizona

[5] Nathan J. Williams, *An Implementation of Scheduler Activations on the NetBSD Operating System*, in: Proceedings of the FREENIX Track, 2002 Usenix Annual Technical Conference, Monterey, CA, USA, http://www.usenix.org/events/usenix02/tech/freenix/-williams.html

[6] Ignatios Souvatzis, *A machine=independent port of the SR language run time system to NetBSD*, in: Proceedings of the 3rd European BSD Conference, Karlsruhe 2004, http://www.arXiv.org/abs/cs.DC/0411028

Fragen?