

Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

IX. Kommunikationsabstraktionen

Wolfgang Schröder-Preikschat

21. Juni 2017



Rekapitulation

Blockierende vs. nichtblockierende Kommunikation

- die Prozessblockade synchronisiert auf die Betriebsmittelbereitstellung
 - Sender ■ benötigt wiederverwendbare Betriebsmittel
 - synchrone IPC ⇒ Nachrichtenplatzhalter ↔ Ziel
 - asynchrone IPC ⇒ Zwischenpuffer
 - ⇒ Nachrichten(platzhalter)deskriptor
 - Empfänger ■ benötigt konsumier- und wiederverwendbare Betriebsmittel
 - synchrone IPC ⇒ Nachricht ↔ Quelle
 - asynchrone IPC ⇒ Zwischenpuffer
 - ⇒ Nachrichten(platzhalter)deskriptor
- asynchrone IPC bedeutet nicht nichtblockierende Kommunikation !!!
 - es meint, Sende- und Empfangsprozess logisch voneinander zu entkoppeln
- „echt nichtblockierend“ bedeutet, send/receive scheitern zu lassen



Gliederung

Einleitung

Abstraktionen

Betriebsmittel

Ausgangssubstanz

Nachrichtenversenden

Speicherdirektzugriff

Sonstiges

Zusammenfassung



Gliederung

Einleitung

Abstraktionen

Betriebsmittel

Ausgangssubstanz

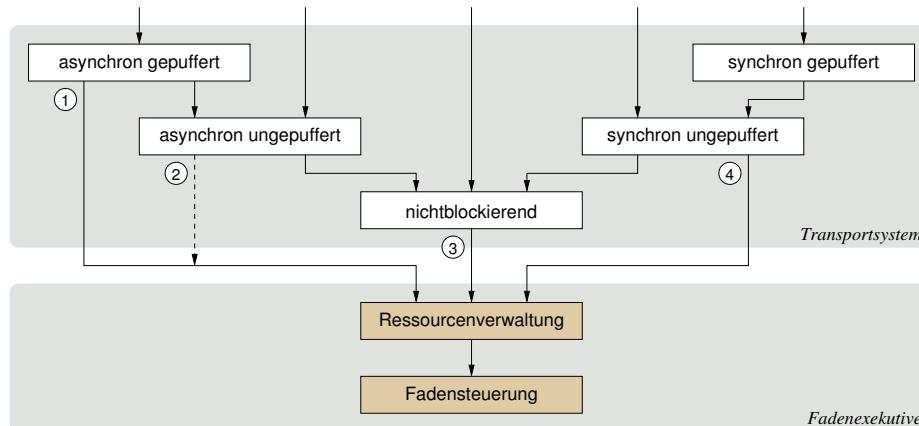
Nachrichtenversenden

Speicherdirektzugriff

Sonstiges

Zusammenfassung





- Schnittstelle zur **Fadenexecutive** (Ressourcenverwaltung):
 - Bereitstellung eines Puffers zur Nachrichtenaufnahme abwarten
 - Bereitstellung eines Pufferdeskriptors zur Nachrichtenerfassung abwarten
 - Verfügbarkeit eines Puffers/Pufferdeskriptors anzeigen
 - Übernahme/Empfang einer Nachricht anzeigen



Synchrone Kommunikation

- braucht Puffer als Betriebsmittel im Anwendungssystem:
 - gepuffert**
 - nach Übergabe an das Transportsystem kann der belegte Speicherbereich sofort wiederverwendet werden
 - und zwar von einem anderen Faden desselben Programms
 - das Programm kann problemlos ausgelagert werden
 - ungepuffert**
 - nach Übergabe an das Transportsystem kann der Transfer der Nachricht End-zu-End stattfinden
 - direkt zwischen Sende- und Empfangsadressraum
 - das Programm kann nur bedingt ausgelagert werden
- beide Verfahren wirken **blockierend** auf den Sendeprozess:
 - die Operation kann nicht wegen Betriebsmittelmangel scheitern
 - da der Sendeprozess zu einem Zeitpunkt niemals mehr als einen Kommunikationsvorgang auslösen kann
 - sie kann jedoch scheitern, wenn der Empfangsprozess ungültig ist



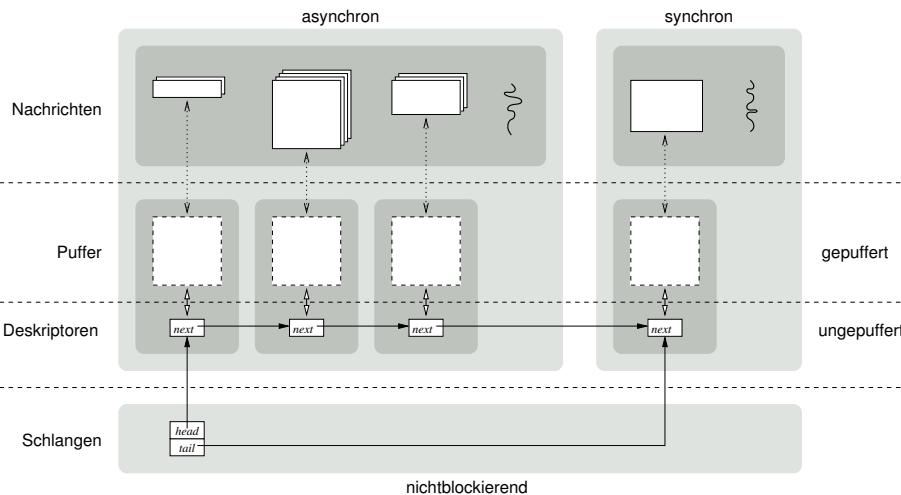
- braucht Puffer/-deskriptoren als Betriebsmittel im Betriebssystem:
 - gepuffert**
 - nach Übergabe an das Transportsystem kann der belegte Speicherbereich sofort wiederverwendet werden
 - durch den Einsatz von **Wechselpufferverfahren** kann der Kopieraufwand vermieden werden
 - ungepuffert**
 - nach Übergabe an das Transportsystem darf der belegte Speicherbereich nur bedingt wiederverwendet werden
 - durch **Signalisierung** des Sendeprozesses wird die mögliche Wiederverwendung des Speicherbereichs angezeigt
- beide Verfahren wirken **blockierend** auf den Sendeprozess, wenn:
 - i die Nachrichtenpuffer oder -deskriptoren ausgegangen sind *und*
 - ii die durch **Betriebsmittelmangel** entstehende Ausnahmesituation nicht zum Scheitern der Operation führen soll



Nichtblockierende Kommunikation

- alle erforderlichen Betriebsmittel werden „von oben“ geliefert:
 - Puffer**
 - von der Anwendungsebene *oder*
 - von der höheren Systemebene
 - die gepufferte Kommunikation implementiert
 - Pufferdeskriptor**
 - von der höheren Systemebene
 - die ungepufferte Kommunikation implementiert
- die **Kommunikationsbetriebsmittel** werden lediglich „delegiert“
 - sie werden Verarbeitungseinheiten (Protokollmaschine, Treiber) zugeführt
 - ihre Freigabe wird den Prozessen signalisiert
- den Prozessen obliegt es, ggf. auf **Freigabesignale** „oben“ zu warten
 - auf der Anwendungs- oder Systemebene





- in Ermangelung von Betriebsmitteln werden Prozesse blockieren
 - hier: Nachrichtenpuffer und -deskriptoren
 - unabhängig von den Synchronität der Kommunikation

Abstraktion von Datenlisten II

Schlangen: Verkettungskopf

```

1 class queue : public chain {
2 protected:
3     chain* last;
4 public:
5     queue () { last = this; }
6     chain* head () { return link; }
7     chain* tail () { return last; }
8
9     void aback (chain* item) {
10        last->next(item);
11        last = item;
12    }
13
14     chain* clear () {
15        chain* item = link;
16        if (item && !(link = item->next())) last = this;
17        return item;
18    }
19 };

```

- minimale Basis für eine (beliebige) dynamische Datenstruktur

```

1 class chain {
2 protected:
3     chain* link;
4 public:
5     chain () { link = 0; }
6     chain* next () { return link; }
7     void next (chain* item) { link = item; }
8 };

```

- Spezialisierung leicht durch eine Schablone (*template*) möglich

```

1 template<typename T>
2 class entry: public chain {
3     T item;
4 public:
5     entry<T>* next () { return link; }
6     void next (entry<T>* item) { link = item; }
7 };

```

Abstraktion von Kommunikationsendpunkten

- Nachrichtenlokalität im Sender-/Empfängerprozess

```
1 typedef long site; // structured endpoint identifier
```

- problemspezifische Identifizierung von „Kommunikationsentitäten“:
Prozesskontrollblock

- direkte Adresse des Sende-/Empfangsprozesses
- Deskriptor einer Prozessinkarnation
- indirekte Adresse des Sende-/Empfangsprozesses
- Zugang (*port*, [2]) zu einer Prozessinkarnation
- Adressierung eines Behältnisses für Nachrichten
- Aus- oder Eingang (*mailbox*)
- Adresse der Verarbeitungsroutine der Nachricht
- Nachrichtenextraktion/-integration [13, S. 256]

- das Bezugssystem bestimmt Struktur und Größe des Bezeichners
 - ähnlich, wie eine MMU Struktur in einer logischen Adresse interpretiert
 - in vernetzten Systemen müssen „Ortskoordinaten“ enthalten sein
 - lokal („vor Ort“) muss die Entität unterscheidbar referenziert werden

■ Nachrichtendescriptor einfach (minimale Basis)

```

1 class alert;           // transfer-event signalling
2
3 struct section {
4     union {
5         void* data; // address as data pointer
6         alert* sign; // address as transfer event
7     } base;
8     unsigned size; // number of bytes at that address
9 };

```

- Quell-/Zieladresse im logischen Adressraum ist auch „Ereignisname“
 - bezeichnet eindeutig das Transfereignis für Sender bzw. Empfänger
- implizite Signalisierung mit Beendigung der Transferoperation (vgl. S. 19)

■ Nachrichtendescriptor verkettbar (minimale Erweiterung)

```
1 class notice: public chain, public section {};
```



Abstraktion zum Botschaftenaustausch I

■ Versenden und Empfangen von Nachrichtensegmenten

```

1 class message : public section {
2 public:
3     int send (site); // no-wait send, unbuffered
4     site receive ();
5 };

```

■ Versenden und Empfangen von Nachrichtenpaketen

```

1 class parcel : public pack {
2 public:
3     site send (site); // synchronization send
4     site receive ();
5 };
6
7 class datagram : public pack {
8 public:
9     int send (site); // no-wait send
10    site receive ();
11 };

```



■ Nachrichtenpaket

```

1 #define PACK_SIZE 64           // beware of the cache
2 #define PART_COUNT (PACK_SIZE / sizeof(section))
3
4 struct pack { // beware of the cache: packet alignment
5     union {
6         char data[PACK_SIZE]; // bounded dataset
7         section part[PART_COUNT]; // descriptor set
8     } sort;
9 };

```

- Platzhalter für einen Datensatz oder mehrere Nachrichtendeskriptoren
- weitere Varianten ergeben sich ggf. aus dem jeweiligen Anwendungsfall
 - wenn z.B. System(prozess)aufrufe als [Fernauftrag](#) [11] laufen sollen

- die Größe ergibt sich auch durch das [Programmiermodell](#) der CPU
 - bei Registernutzung zum Botschaftenaustausch zwischen Prozessen *oder*
 - der Ausnutzung der Zwischenspeicherzeile (*cache line*) zum Datentransfer



Abstraktion zum Botschaftenaustausch II

■ Versenden, Empfangen, Beantworten von Anforderungsnachrichten

```

1 class request : public pack {
2 public:
3     site send (site, pack&); // remote-invocation send
4     site receive ();
5     int reply (site);
6
7     site send(site peer) {
8         return send(peer, *this);
9     }
10 };

```

■ Verarbeitungsschleife von Anforderungsnachrichten

```

1 for (;;) {
2     request msg;
3     site peer = msg.receive();
4     // carry out order as encoded in the request
5     msg.reply(peer);
6 }

```



■ Auslösung adressraumübergreifender Datentransfers

```

1 class dataset : public section {
2 public:
3     // various dataset constructors come here...
4     int fetch (site, section&); // copy from peer
5     int store (site, section&); // copy to peer
6 };

```

■ Verwendungsmuster der Transferoperationen

```

1 dataset from, to;
2
3 from.fetch(peer, to);    // copy data from peer to me
4 to.store(peer, from);   // copy data from me to peer

```

- section-Attribute des dataset sind mittels pack kommunizierbar
 - z.B. über die Verarbeitungsschleife von Anforderungsnachrichten (S. 16)
- ohne Zwischenpufferung wird zwischen den Adressräumen transferiert



Abstraktion zum Datentransfer III „zählender Fadensemaphor“

■ Signalisierung von Transfereignissen

```

1 class alert {
2 public:
3     int raise (int);        // simulate fetch/store event
4     int await (unsigned);   // block on fetch/store event
5 };

```

- ein Faden kann ein fetch/store auf seinen Adressräumen erfassen
 - signalisiert wird die Basisadresse des zugegriffenen Adressraumabschnitts
 - diese Adresse ist (in dem Fall) auch die Adresse einer alert-Inkarnation
 - angezeigt wird Konsumier-/Wiederverwendbarkeit eines Betriebsmittels
 - store** liefert das konsumierbare Betriebsmittel „Nachricht“
→ **await** liefert die Anzahl der transferierten Bytes
 - fetch** macht das wiederverwendbare Betriebsmittel „capture“ frei
→ **await** liefert 0

- der Faden wartet eine gegebene Anzahl solcher Ereignisse ab
 - ein zählender Semaphor ist Komponente des Fadenkontrollblocks



■ Auslösung adressraumübergreifender, signalisierender Datentransfers

```

1 class capture : public dataset {
2 public:
3     // various capture constructors come here...
4     int fetch (site, section&); // copy from peer & alert
5     int store (site, section&); // copy to peer & alert
6 };

```

■ Verwendungsmuster der Transferoperationen

```

1 capture from, to;
2
3 from.fetch(peer, to);    // signal peer when fetched
4 to.store(peer, from);   // signal peer when stored

```

- Ereignis {from,to}.base.sign (S. 13) wird im peer-Kontext angezeigt
- ein auf dieses Ereignis wartender Faden im peer-Team wird freigestellt
- das peer-Team erfährt von Lese-/Schreibzugriffen auf seinen Adressraum
 - genauer: von der Beendigung der korrespondierenden Transferoperationen



Fadensemaphor

Semaphor [5], der fest mit einem Faden verbunden ist und ansonsten die üblichen zählenden Eigenschaften besitzt

- der Semaphor ist **Attribut** des Fadendeskriptors bzw. -kontrollblocks
 - der Fadendeskriptor enthält, erbt, „benutzt“ einen Semaphordeskriptor
 - über diesen werden **fadenspezifische Ereignisse** signalisiert und verwaltet
- seine Bedeutung im Zusammenhang mit asynchronen Datentransfers:
 - Anzeige der Konsumier-/Wiederverwendbarkeit eines Betriebsmittels
 - Synchronisierung mit vielen Lesern/Schreibern desselben Speicherbereichs
 - Unterstützung von asynchronen, prozessübergreifenden Prozeduraufrufen
 - mit einem „Versprechen“ (*promise*, [6, 10]) zur Ergebnislieferung
 - die „letztendlich“ (*eventual*, [7]) beim aufrufenden Prozess eintrifft und
 - Erwartungen an eigene Berechnungen in der „Zukunft“ (*future*, [1]) weckt
 - Kommunikation „gleichgetakteter“ Prozesse nach dem SPMD [4] Modell
- in Anlehnung an den PEACE „Segmenttransfer“ [12, S. 370 ff.]



■ Versenden einer Nachricht ungepuffert im Transportsystem

```
1 int send (site peer, message& msg) {  
2     msg.send(peer);           // pass descriptor  
3     return msg.base.sign->await(1);  
4 }
```

`send` versendet den Nachrichtendeskriptor nichtwartend (*no-wait*)
`await` blockiert den Faden bis zur Beendigung von `fetch`

■ Empfangen einer Nachricht gepuffert im Anwendungssystem

```
1 int receive (section& buf) {  
2     buf.base.data = 0;          // nothing received yet  
3  
4     message msg;  
5     site peer = msg.receive(); // fill descriptor  
6     if (peer && (buf.base.data = new char[msg.size]))  
7         buf.size = capture(msg).fetch(peer, buf);  
8 }
```

Gliederung

Einleitung

Abstraktionen

- Betriebsmittel
- Ausgangssubstanz
- Nachrichtenversenden
- Speicherdirektzugriff
- Sonstiges

Zusammenfassung

Mikrokerne in Reinform [9] sind in ihrer Leistungsfähigkeit parallelen Systemen eher abträglich \sim „*microkernel considered harmful*“

- hier war und ist schon immer prozessorübergreifende IPC Normalität
 - insb. auf `verteilten Speicher` (*distributed memory*) basierende Systeme
 - vorrangig ist die `minimale Rüstzeit` (*setup time*) für die Kommunikation
 - dabei ist lokale IPC die Ausnahme, aber `globale IPC` ein Muss
- Optimierungsmaßnahmen für lokale Operationen verlieren an Gewicht
 - System- und Anwendungsprozesse residieren eher selten am selben Ort
 - Prozessorregister als Nachrichtenbehälter [3, 8] sind wenig wirkungsvoll

Vielkerner (*many-core processor*) lassen die kooperative Einplanung von Fäden kernlokal immer mehr zur Normalität werden

- das impliziert jedoch nicht die kernlokale IPC zwischen den Fäden
- vielmehr ist auf die Zwischenspeicherzeile (*cache line*) zu achten
 - die nicht nur Größe und Ausrichtung der Nachrichtenpuffer bestimmt
 - sondern überhaupt die Implementierung von IPC querschneidend belangt

Resümee

- Rekapitulation: blockierende vs. nichtblockierende Kommunikation
 - asynchrone IPC bedeutet nicht nichtblockierende Kommunikation
 - „echt nichtblockierend“ bedeutet, `send/receive` scheitern zu lassen
- Zusammenhang von Kommunikationsart und Betriebsmittel
 - asynchrone Kommunikation, gepuffert/ungepuffert
 - synchrone Kommunikation, gepuffert/ungepuffert
 - nichtblockierende Kommunikation \sim Betriebsmittel delegieren
- Kommunikationsabstraktionen in Anlehnung an PEACE [12]
 - Datenlisten: Verkettungsglied und -kopf, Spezialisierung
 - Kommunikationsendpunkte: Nachrichtenlokalität (Sender/Empfänger)
 - Nachrichtenstrukturen: Nachrichtendeskriptor und -paket
 - Botschaftenaustausch: *no-wait, synchronization, remote invocation*
 - Datentransfer: adressraumübergreifend (DMA), signalisierend
 - Fadensemaphor: *promise, eventual, future*
- Konzepte, die logisch systemweit (kern-/netzübergreifend) greifen

Literaturverzeichnis I

- [1] BAKER, JR., H. C. ; HEWITT, C. :
The Incremental Garbage Collection of Processes.
In: LOW, J. (Hrsg.): *Proceedings of the 1977 ACM Symposium on Artificial Intelligence and Programming Languages (AIPL '77)*, ACM, 1977, S. 55–59
- [2] BALZER, R. M. :
PORTS—A Method for Dynamic Interprogram Communication and Job Control.
In: *Proceedings of the Spring Joint Computer Conference (AFIPS'71)* Bd. 38
American Federation of Information Processing Societies, 1971, S. 485–489
- [3] CHERITON, D. R.:
An Experiment Using Registers for Fast Message-Based Interprocess Communication.
In: *SIGOPS Operating Systems Review* 18 (1984), Okt., Nr. 4, S. 12–20
- [4] DAREMA-ROGERS, F. ; GEORGE, D. A. ; NORTON, V. A. ; PFISTER, G. F.:
A VM Parallel Environment / IBM.
1985 (RC 11225). –
IBM Research Report



Literaturverzeichnis III

- [9] LIEDTKE, J. :
On μ -Kernel Construction.
In: JONES, M. B. (Hrsg.): *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, ACM Press, 1995. –
ISBN 0-89791-715-4, S. 237–250
- [10] LISKOV, B. J. H. ; SHIRIRA, L. :
Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.
In: WEXELBLAT, R. L. (Hrsg.): *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, ACM Press, 1988. –
ISBN 0-89791-269-1, S. 260–267
- [11] NELSON, B. J.:
Remote Procedure Call.
Pittsburg, PA, USA, Department of Computer Science, Carnegie-Mellon University,
Diss., Mai 1981
- [12] SCHRÖDER-PREIKSCHAT, W. :
The Logical Design of Parallel Operating Systems.
Upper Saddle River, NJ, USA : Prentice Hall International, 1994. –
ISBN 0-13-183369-3



Literaturverzeichnis II

- [5] DIJKSTRA, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)
- [6] FRIEDMAN, D. P. ; WISE, D. S.:
The Impact of Applicative Programming on Multiprocessing.
In: *Proceedings of the 1977 International Conference on Parallel Processing*, IEEE Computer Society, 1977, S. 263–272
- [7] HIBBARD, P. :
Parallel Processing Facilities.
In: SCHUMAN, S. A. (Hrsg.): *New Directions in Algorithmic Languages*.
Institut de Recherche en Informatique et Automatique (IRIA), 1976, S. 1–7
- [8] LIEDTKE, J. :
Improving IPC by Kernel Design.
In: BLACK, A. P. (Hrsg.) ; LISKOV, B. J. H. (Hrsg.): *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP '93)*, ACM, 1993. –
ISBN 0-89791-632-8, S. 175–188



Literaturverzeichnis IV

- [13] VON EICKEN, T. ; CULLER, D. E. ; GOLDSTEIN, S. C. ; SCHAUER, K. E.:
Active Messages: a Mechanism for Integrated Communication and Computation.
In: GOTTLIEB, A. (Hrsg.): *Proceedings of the 19th International Symposium on Computer Architecture (ISCA '92)*, ACM, 1992. –
ISBN 0-89791-509-7, S. 256–266

