

D Überblick über die 3. Übung

D Überblick über die 3. Übung

- Infos zur Aufgabe2: Abstraktionen für das Kommunikationssystem
- Wiederholung Sockets
- Wiederholung Threads
 - ◆ Pthreads
- Organisatorisches
 - ◆ Übungsbetrieb mit den Lego Mindstorms RCX
- Programmieren auf dem Lego Mindstorms RCX

D.1 eine einfache Kommunikationsschnittstelle

D.1 Aufgabe2

- einheitliche Kommunikationsschnittstellen:

```
class CommunicationSystem {
public:
    CommunicationSystem(Address addr);
    virtual Address getLocalAddress() = 0;

    virtual bool send( Address dest,
                      Buffer *message,
                      int len) = 0;
    virtual bool receive( Address *src,
                         Buffer *buffer)=0;
};
```

```
class Buffer { public: char *buffer; int len; };
```

- folgende Kommunikation soll unterstützt werden:

- ◆ Unix <-> Unix
- ◆ Unix <-> RCXSimulator
- ◆ Linux <-> RCX

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2006

SocketsThreadsRCX.fm 2006-05-24 14.09

D.1

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2006

SocketsThreadsRCX.fm 2006-05-24 14.09

D.3

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

D.1 Aufgabe2

D.1 Aufgabe2

- zählende Semaphore

```
class Semaphor{
public:
    Semaphor(int startvalue);
    void P(void);
    void V(void);
};
```

- einheitliche Schnittstelle zur Threadverwaltung

```
class Runnable {
    virtual void run() = 0;
};

class Thread {
public:
    Thread(Runnable *run);
};
```

D.2 UDP Kommunikation

D.2 UDP Kommunikation

- Sockets (Wiederholung)
 - ◆ Erzeugung
 - ◆ Binden
 - ◆ Socket Adressen
- UDP-Sockets
 - ◆ Datagramme senden / empfangen

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2006

SocketsThreadsRCX.fm 2006-05-24 14.09

D.2

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2006

SocketsThreadsRCX.fm 2006-05-24 14.09

D.4

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Erzeugen eines neuen Sockets:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- **domain** gibt die Kommunikationsdomäne an, z.B.
 - ◆ **PF_INET**: Internet, IPv4
 - ◆ **PF_UNIX**: Unix Filesystem, lokale Kommunikation
 - ◆ **PF_APPLETALK**: Appletalk Netzwerk
- Durch **type** wird die Kommunikationssemantik festgelegt
z.B.: in **PF_INET** Domain:
 - ◆ **SOCK_STREAM**: Stream-Socket
 - ◆ **SOCK_DGRAM**: Datagramm-Socket
 - ◆ **SOCK_RAW**

VS - Übung

1 Erzeugen eines neuen Sockets (2)

- **protocol** legt das Protokoll fest.
 - ◆ 0 bedeutet hierbei: Standardprotokoll für Domain/Type Kombination
 - ◆ Zu jeder Kombination aus Sockettyp/-familie gibt es aktuell nur ein Protokoll:
 - **PF_INET, SOCK_STREAM**: → TCP
 - **PF_INET, SOCK_DGRAM**: → UDP

VS - Übung

2 Binden von Sockets

- **bind** bindet einen Socket an eine lokale IP-Adresse und Port:

```
int bind(int sockfd,
        struct sockaddr *my_addr, socklen_t addrlen);
```

- ◆ **sockfd**: Socketdeskriptor
- ◆ **my_addr**: Protokollspezifische Adresse
(Address Family AF_INET, IP-Adresse, Port)
- ◆ **addrlen**: Größe der Adresse in Byte

VS - Übung

3 Socket Adressen

- Socket-Interface (**<sys/socket.h>**) ist protokoll-unabhängig

```
struct sockaddr {
    sa_family_t    sa_family;    /* Adressfamilie */
    char           sa_data[14];  /* Adresse */
};
```

- Internet-Protokoll-Familie (**<netinet/in.h>**) verwenden

```
struct sockaddr_in {
    sa_family_t    sin_family;    /* = AF_INET */
    in_port_t      sin_port;      /* Port */
    struct in_addr sin_addr;      /* Internet-Adresse */
    char           sin_zero[8];   /* Füllbytes */
};
```

VS - Übung

4 Lokales Binden eines Sockets

- **INADDR_ANY**: wenn Socket auf allen lokalen Adressen (z.B. allen Netzwerkinterfaces) Verbindungen akzeptieren soll
- **sin_port = 0**: Portnummer wird vom System ausgewählt
- Adresse und Port müssen in Netzwerk-Byteorder vorliegen
- Beispiel

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(MYPORT);
...
s = socket(PF_INET, SOCK_DGRAM, 0);
bind(s, (struct sockaddr *) &my_addr, sizeof(my_addr));
```

5 Socket-Adresse aus Hostnamen erzeugen

- **gethostbyname** liefert Informationen über einen Host

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

- Struktur **hostent**

```
struct hostent {
    char*   h_name;           /* offizieller Rechnername */
    char**  h_aliases;        /* alternative Namen */
    int     h_addrtype;       /* == AF_INET */
    int     h_length;         /* Länge einer Adresse */
    char**  h_addr_list;      /* Liste von Netzwerk-Adressen,
                               Abgeschlossen durch NULL */
};

#define h_addr h_addr_list[0]
```

5 Socket-Adresse aus Hostnamen erzeugen (2)

- Beispiel

```
char *hostname = "fau40";
int port = 4711;
struct hostent *host;
struct sockaddr_in saddr;

if ( (host = gethostbyname(hostname)) == NULL){
    perror("gethostbyname()");
    exit(EXIT_FAILURE);
}

memset(&saddr, 0, sizeof(saddr)); /* initialisieren */

/* Adresse kopieren*/
memcpy((char *) &saddr.sin_addr,
       (char *) host->h_addr, host->h_length);

/* Protokoll und Port festlegen*/
saddr.sin_family = AF_INET;
saddr.sin_port = htons(port);

/* saddr verwenden ... z.B. bind, sendto*/
```

6 Schließen einer Socketverbindung

- **close**: schließen des Socketdeskriptors

```
#include <unistd.h>

int close(int s);
```

- **shutdown**: schließen der Verbindung

```
int shutdown(int s, int how);
```

- ◆ **how**:

- SHUT_RD: verbiete Empfang
- SHUT_WR: verbiete Senden
- SHUT_RDWR: verbiete Senden und Empfangen

7 UDP: Senden

- Verbindungslos:
Empfänger wird bei jedem Paket angegeben

- Senden

```
int sendto(int s, const void *buf, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

- ◆ **s**: Socketdeskriptor
- ◆ **buf**: Zeiger auf Daten
- ◆ **len**: Länge der Daten (Fehler, wenn zu groß für ein UDP Paket)
- ◆ **flags**: weitere Parameter (z.B. DONTWAIT)
- ◆ **to**: Zieladresse
- ◆ **tolen**: Länge der Zieladresse

8 UDP: Empfangen

- Verbindungslos:
Absender kann aus dem empfangenen Paket gelesen werden

- Empfangen

```
int recvfrom(int s, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

- ◆ **s**: Socketdeskriptor
- ◆ **buf**: Puffer um die Daten aufzunehmen
- ◆ **len**: Größe des Puffers
- ◆ **flags**: Parameter (z.B. MSG_PEEK)
- ◆ **from**: Zeiger auf Adressstruktur um die Absenderadresse aufzunehmen
- ◆ **fromlen**: Länge der Adressstruktur bei Aufruf
enthält nach dem Aufruf die Länge der eingefügten Adresse

9 Lesen und Schreiben auf UDP Sockets - Beispiel

- Beispiel: Server, der alle Eingaben wieder zurückschickt

```
int fd = socket(PF_INET, SOCK_DGRAM, 0); /* Fehlerabfrage */
struct sockaddr_in name;
/* ... */
bind(fd, (const struct sockaddr *)&name, sizeof(name)); /* ... */

for(;;) {
    char buf[70000];
    struct sockaddr_in addr;
    socklen_t addr_sz = sizeof(addr);
    int rcv_len;

    // empfangen
    rcv_len = recvfrom(fd, buf, sizeof(buf), 0,
                      (struct sockaddr *)&addr, &addr_sz);
    /* Fehlerabfrage */

    // und wieder zurück
    sendto(fd, buf, rcv_len, 0,
           (const struct sockaddr *)&addr, sizeof(addr))
    /* Fehlerabfrage */
}
close(fd);
```

D.3 Threads

- Vergleich von Prozess und Thread-Konzepten
- POSIX-Threads

1 Motivation von Threads

- UNIX-Prozesskonzept ist für viele heutige Anwendungen unzureichend
- in Multiprozessorsystemen werden häufig parallele Abläufe in einem virtuellen Adreßraum benötigt
- zur besseren Strukturierung von Problemlösungen sind oft mehrere Aktivitätsträger innerhalb eines Adreßraums nützlich
- typische UNIX-Server-Implementierungen benutzen die **fork**-Operation, um einen Server für jeden Client zu erzeugen
 - ➔ Verbrauch unnötig vieler System-Ressourcen (Datei-Deskriptoren, Page-Table, Speicher, ...)

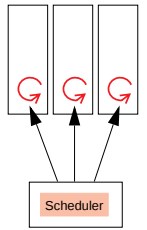
VS - Übung

2 Vergleich von Prozess- und Thread-Konzepten (2)

- mehrere **UNIX-Prozesse** mit gemeinsamen Speicherbereichen

Bewertung:

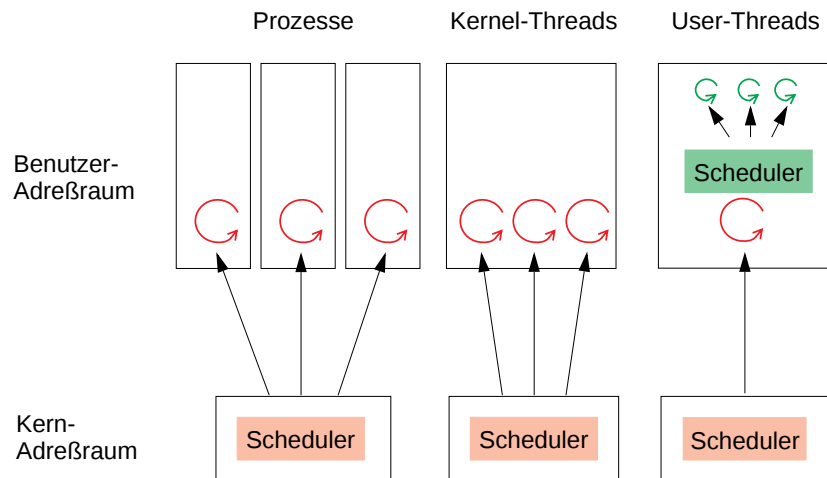
- + echte Parallelität möglich
- viele Betriebsmittel zur Verwaltung eines Prozesses notwendig; Prozessumschaltungen aufwendig → teuer
- innerhalb einer solchen Prozessfamilie wäre häufig ein anwendungsorientiertes Scheduling notwendig: schwierig realisierbar



VS - Übung

1 Vergleich von Prozess- und Thread-Konzepten

D.3 Threads



VS - Übung

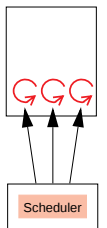
2 Vergleich von Prozess- und Thread-Konzepten (3)

D.3 Threads

- **Kernel-Threads:** leichtgewichtige Prozesse (*lightweight processes*)

Bewertung:

- + gemeinsame Nutzung von Betriebsmitteln
- + jeder leichtgewichtige Prozess ist dem Betriebssystemkern bekannt
 - eigener Programmzähler, Registersatz, Stack
- + Umschalten zwischen zwei leichtgewichtigen Prozessen einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung
 - ➔ es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf)
 - ➔ Adreßraum muss nicht gewechselt werden
 - ➔ alle Systemressourcen bleiben verfügbar
- Verwaltung und Scheduling meist durch Kern vorgegeben



VS - Übung

2 Vergleich von Prozess- und Thread-Konzepten (4)

- **User-Level-Threads** (Koroutinen) — Realisierung von Threads auf Benutzerebene innerhalb eines Prozesses

Bewertung:

- + Erzeugung von Threads und Umschaltung extrem billig
- + Verwaltung und Scheduling anwendungsorientiert möglich
- Systemkern hat kein Wissen über diese Threads
 - ➔ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)
 - ➔ in Multiprozessorsystemen keine parallelen Abläufe möglich
 - ➔ wird eine Koroutine wegen eines *page faults* oder in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert



3 UNIX — Prozesse, LWPs & Threads

- Thread-Konzept zunehmend auch in UNIX-Systemen realisiert
 - ◆ Solaris
 - ◆ HP UX
 - ◆ Digital UNIX
 - ◆ Linux
 - ◆ ...
- Programmierschnittstelle standardisiert: **Pthreads-Bibliothek**
 - ➔ IEEE POSIX Standard P1003.4a
- Pthreads-Implementierungen aber sehr unterschiedlich!
 - reine User-level-Threads (HP-UX)
 - reine Kernel-Threads (Linux, MACH, KSR-UNIX, Digital UNIX)
 - parametrierbare Mischung (Solaris)
- Daneben z. T. auch andere Thread-Bibliotheken (z. B. Solaris-Threads)

2 Vergleich von Prozess- und Thread-Konzepten (5)

- Vergleich

	Prozesse	Kernel-Threads	User-Threads
Kosten	– teuer	O mittel	+ billig
Betriebssystemeingliederung	+ gut	+ gut	– schlecht
Interaktion untereinander	– schwierig	+ einfach	+ einfach
Benutzerkonfigurierbarkeit	– nein	– nein	+ ja
Gerechtigkeit	– nein	+ ja	± teils

- Gerechtigkeit bedeutet:
wie kommt das System damit klar, wenn eine Anwendung eine große Anzahl von Aktivitätsträgern erzeugt, eine andere dagegen eine geringe — werden Zeitscheiben an Anwendungen oder an Aktivitätsträger vergeben?

4 pthread-Benutzerschnittstelle

- Pthreads-Schnittstelle (Basisfunktionen):

<i>pthread_create</i>	Thread erzeugen & Startfunktion angeben
<i>pthread_exit</i>	Thread beendet sich selbst
<i>pthread_join</i>	Auf Ende eines anderen Threads warten
<i>pthread_self</i>	Eigene Thread-Id abfragen
<i>pthread_yield</i>	Prozessor zugunsten eines anderen Threads aufgeben

4 pthread-Benutzerschnittstelle (2)

■ Threaderzeugung

```
#include <pthread.h>
/*Compilieren mit -lpthread*/

int pthread_create( pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg)
```

- ◆ **thread** Thread-ID
- ◆ **attr** modifizieren von Attributen des erzeugten Threads (z. B. Stackgröße). **NULL** für Standardattribute.
- ◆ Thread wird erzeugt und startet mit der Ausführung der Funktion **start_routine** mit Parameter **arg**
- ◆ Rückgabewert: 0; im Fehlerfall -1 außerdem wird **errno** gesetzt

4 pthread-Benutzerschnittstelle (3)

■ explizites beenden eines Threads:

```
void pthread_exit(void *retval)
```

- ◆ Der Thread wird beendet und **retval** wird als Rückgabewert zurück geliefert (siehe **pthread_join**)

■ Auf Thread warten und exit-Status abfragen:

```
int pthread_join(pthread_t thread, void **retvalp)
```

- ◆ Wartet auf den Thread mit der Thread-ID **thread** und liefert dessen Rückgabewert über **retvalp** zurück.

5 Beispiel (Multiplikation Matrix mit Vektor)

```
double a[100][100], b[100], c[100];
int main(int argc, char* argv[]) {
    pthread_t tids[100];
    ...
    for (i = 0; i < 100; i++)
        pthread_create(&tids[i], NULL, mult, (void *)i);
    for (i = 0; i < 100; i++)
        pthread_join(tids[i], NULL);
    ...
}

void *mult(void *cp) {
    int j, i = (int)cp;
    double sum = 0;

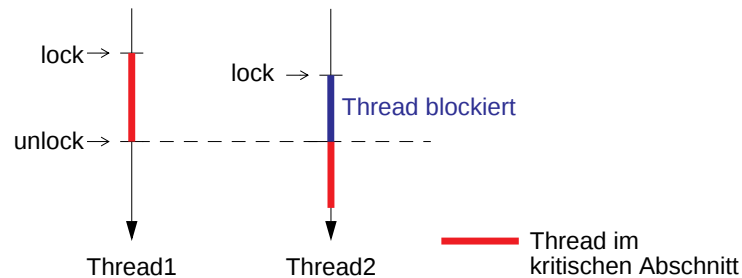
    for (j = 0; j < 100; j++)
        sum += a[i][j] * b[j];
    c[i] = sum;
    return 0;
}
```

6 Pthreads-Koordinierung

- UNIX-Semaphore für Koordinierung von leichtgewichtigen Prozessen zu teuer
 - ◆ Implementierung durch den Systemkern
 - ◆ komplexe Datenstrukturen
- Bei Koordinierung von Threads reichen meist einfache **mutex**-Semaphore
 - ◆ gewartet wird durch Blockieren des Threads oder durch *busy wait* (*Spinlock*)

6 Pthreads-Koordinierung - Mutexes

- Koordinierung von kritischen Abschnitten



VS - Übung

6 Pthreads-Koordinierung - Mutexes (2)

- Mutex erzeugen

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr)
```

- Lock & unlock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Beispiel:

```
pthread_mutex_t m1;
pthread_mutex_init(&m1, NULL);
...
pthread_mutex_lock(&m1);
... kritischer Abschnitt
pthread_mutex_unlock(&m1);
```

VS - Übung

6 Pthreads-Koordinierung - Mutexes (3)

- Komplexere Semaphore können alleine mit Mutexes nicht implementiert werden

➔ Problem:

- Ein Mutex sperrt die Datenstruktur des komplexen Semaphors
- Der Zustand der Datenstruktur erlaubt die Operation nicht
- Blockieren an einem weiteren Mutex kann zu Verklemmungen führen

➔ Lösung: mutex in Verbindung mit sleep/wakeup-Mechanismus

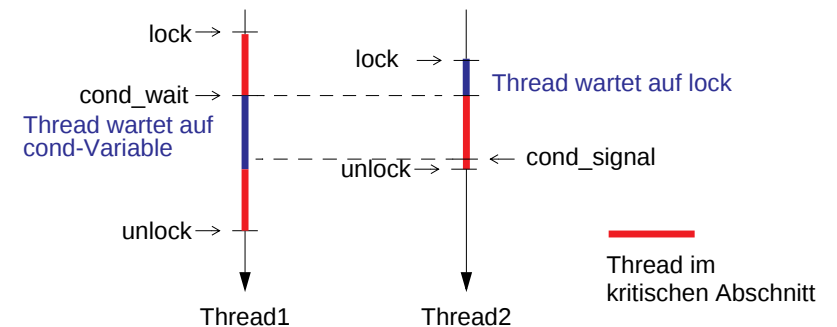
➔ **Condition Variables**

VS - Übung

6 Pthreads-Koordinierung - Condition Variables

- ★ Condition Variables

- Mechanismus zum Blockieren (mit gleichzeitiger Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads



VS - Übung

6 Pthreads-Koordinierung - Condition Variables(2)

D.3 Threads

- Realisierung
 - ◆ Thread reiht sich in Warteschlange der Condition Variablen ein
 - ◆ Thread gibt Mutex frei
 - ◆ Thread gibt Prozessor auf
 - ◆ Ein Thread der die Condition Variable "frei" gibt weckt einen (oder alle) darauf wartenden Threads auf
 - ◆ Deblockierter Thread muß als erstes den kritischen Abschnitt neu betreten (lock)
 - ◆ Da möglicherweise mehrere Threads deblockiert wurden, muß die Bedingung nochmals überprüft werden

6 Pthreads-Koordinierung - Condition Variables(4)

D.3 Threads

- Beispiel:

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
...
/* Betriebsmittel belegen */
pthread_mutex_lock(&m1);
while (resource_busy)
    pthread_cond_wait(&c1, &m1);
resource_busy = TRUE;
pthread_mutex_unlock(&m1);

...
/* Betriebsmittel nutzen */
...

/* Betriebsmittel freigeben */
pthread_mutex_lock(&m1);
resource_busy = FALSE;
pthread_cond_signal(&c1);
pthread_mutex_unlock(&m1);
```

6 Pthreads-Koordinierung - Condition Variables(3)

D.3 Threads

- Condition Variable erzeugen

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

- auf Bedingung warten & Bedingung signalisieren

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- ◆ **pthread_cond_signal**: mindestens ein wartender Thread wird aufgeweckt — es ist allerdings nicht definiert welcher
- ◆ **pthread_cond_broadcast**: alle wartenden Threads werden aufgeweckt
- ◆ Ein aufwachender Thread wird als erstes den Mutex neu belegen — ist dieser gerade gesperrt bleibt der Thread solange blockiert!

7 Zusammenfassung - Pthreads

D.3 Threads

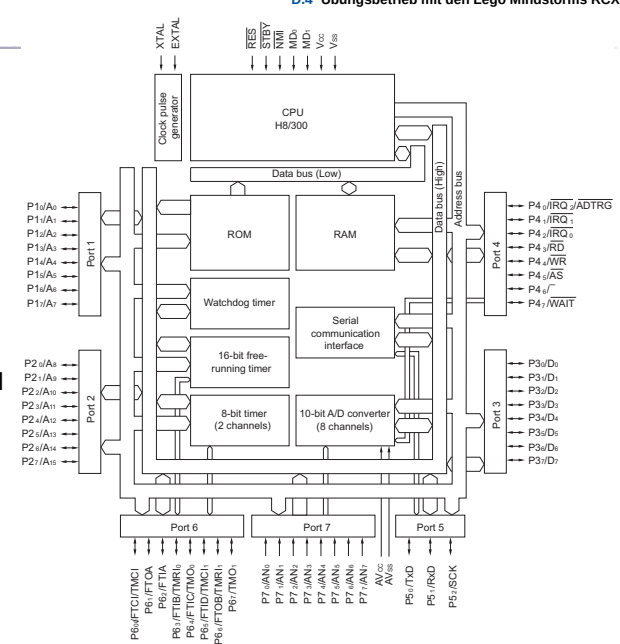
- standardisierte Thread API
- unterschiedliche Implementierung
- einfache Threaderzeugung mittels **pthread_create**
- Koordinierung mit Hilfe von Mutexes
pthread_mutex_lock, **pthread_mutex_unlock**
- Koordinierung mit Hilfe von Condition Variables
pthread_cond_wait,
pthread_cond_signal, **pthread_cond_broadcast**

D.4 Übungsbetrieb mit den Lego Mindstorms RCX

- Mindstorms RIS/RCX
 - ◆ Hardware
 - ◆ Wie erstelle ich ein Programm für den RCX?
- der RCX Simulator
- der echte RCX Baustein (Wo?, Wann?, Wie?)
- unsere API

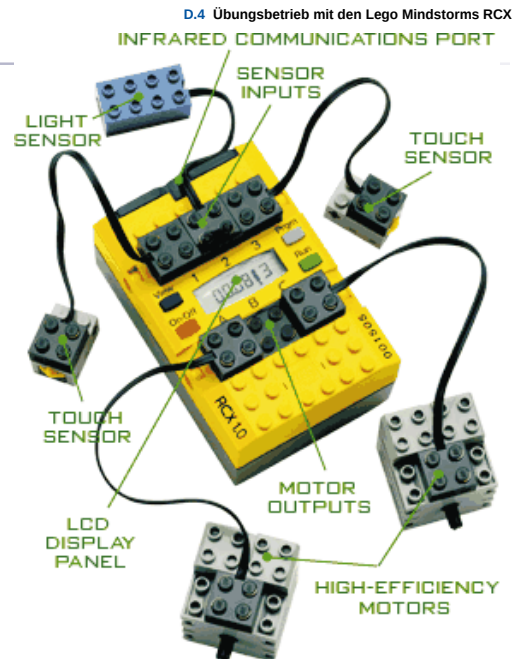
2 H8 / 3292

- Familie H8/3297
- 16 bit Prozessor (H8/300)
- 16 KByte ROM
- 512 Byte int. RAM
- im RCX zusätzl. 32 KByte ext. RAM
- Von-Neuman-Architektur



1 RCX

- liegt dem *Lego Robotics Invention System* bei.
- Interaktion mit der Umwelt:
 - ◆ 3 Sensoren
 - ◆ 3 Aktoren
 - ◆ 4 Knöpfe
 - ◆ 4-5 stelliges Display
 - ◆ IR-Schnittstelle
- enthält einen Hitachi H8/3292 Microcontroller

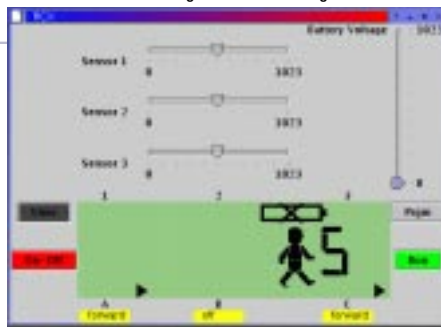


3 Wie schreibe ich ein Programm für den RCX?

- Programm für den RCX erstellen.
 - ◆ compilieren mittels Cross-Compilers für H8/300
`/proj/i4vs/pub/RCX/bin/h8300-e1f-*`
 - ◆ hinzulinken einer Sammlung von Betriebssystemcode
`/proj/i4vs/pub/RCX/bin/kernel.a`
 - ◆ verwenden des passenden Linkerskripts
`/proj/i4vs/pub/RCX/rcx.lids`
 - ◆ konvertieren ins SREC-Format (mittels `h8300-e1f-objcopy`)
- sehr einfaches Beispielprogramm incl. Makefile unter `/proj/i4vs/pub/RCX/rcx_example`
Bitte das Makefile anschauen
- Laden eines Programmes in den RCX mittels Firmware-Downloader
`/proj/i4vs/pub/RCX/bin/firmdl`
- Testen mittels RCXSimulator

4 RCXSimulator

- Simuliert die CPU auf Instruktionsebene
- Unterstützung für den GDB
- Kommunikation wird mittels UDP Pakete simuliert (jedes Paket mit einer Payload von 1 Byte)



- Simulator starten

```
setenv RCXSIMDIR /proj/i4vs/pub/RCX/RCXSimulator
java -jar ${RCXSIMDIR}/RCXSimulator.jar \
  -r ${RCXSIMDIR}/util/srec/LEGO/modROM.srec \
  -ir=UDP:8000 -f my.srec
```

- **-ir=UDP:8000** : setzt den UDP Port der simulierten Kommunikationsschnittstelle
 - ◆ passendes Gegenstück: z.B.


```
java -cp ${RCXSIMDIR}/util/terminal Terminal
```

5 RCXSimulator und der GDB

- Simulator starten mit der Option **-gdb** und **-gdbport 4711**

```
> java -jar RCXSimulator.jar -r modROM.srec -f my.srec \
  -gdb -gdbport 4711
```

```
RCX Simulator - (c) 2000, 2003-2004 mf
.....
waiting for GDB connection on port 4711...
```

- in einer anderen Shell

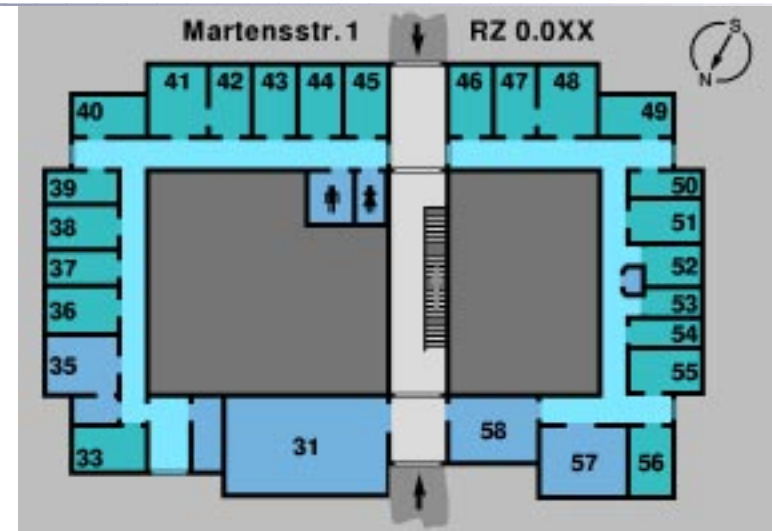
```
> /proj/i4vs/pub/RCX/bin/h8300-elf-gdb my.elf32
GNU gdb 6.1
.....
(gdb) target remote localhost:4711
Remote debugging using localhost:4711
0x00000000 in ?? ()
(gdb) c
Continuing.
```

6 Wo sind die echten RCX Bausteine?

- 3 CIP-Pool Rechner in der Manlobbi (0.058) mit IR Sendern (Tower) **manlobbi-1**, **manlobbi-2**, **manlobbi-3**
- Der IR-Tower ist über **/dev/usb/legousbtower0** zu erreichen
- Übertragen einer Firmware mittels:


```
/proj/i4vs/pub/RCX/bin/firmdl --fast
  --usb --tty=/dev/usb/legousbtower0 my.srec
```
- RCX Bausteine bitte immer bei Hr. Christiani (0.056) abholen und wieder zurückbringen

7 Wo ist die Manlobbi?



8 API auf dem RCX

- ◆ wird evtl. im Laufe der Übung erweitert
- ◆ bietet grundlegende Möglichkeiten den RCX zu verwenden
- ◆ Header: `/proj/i4vs/pub/RCX/api.h`

■ Ausgabe auf dem LC-Display

```
void cputs(char *s);
void cputc(char c, int pos); /* 0 <= pos <= 5*/
```

■ Threads

```
ThreadDesc *thread_create ( void (*thread_start)(void *),
                             void *argv, int stacksize);
```

■ Semaphore

```
/* Semaphore */
void init_semaphore(Semaphore *s, short startvalue);
void p(Semaphore *s);
void v(Semaphore *s);
```

8 API auf dem RCX (2)

■ Kommunikation (serielle Kommunikation über die IR-Schnittstelle)

```
int sendchar(unsigned char c); /* synchron!*/
int rcvchar(void); /* blockierend */

int printf(const char *fmt, ...);
```

■ Kommunikation ist unzuverlässig!