

Übungen zu Systemprogrammierung 2

Ü4 – Thread-Koordinierung

Wintersemester 2025/26

Jürgen Kleinöder, Thomas Preisner, Tobias Häberlein, Ole Wiedemann

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät



- 4.1 Mutexe und Bedingungsvariablen
- 4.2 Nichtblockierende Synchronisation
- 4.3 Module und Symbole
- 4.4 Statische Bibliotheken
- 4.5 Dynamische Bibliotheken
- 4.6 Aufgabe 4: jbuffer



4.1 Mutexe und Bedingungsvariablen

4.2 Nichtblockierende Synchronisation

4.3 Module und Symbole

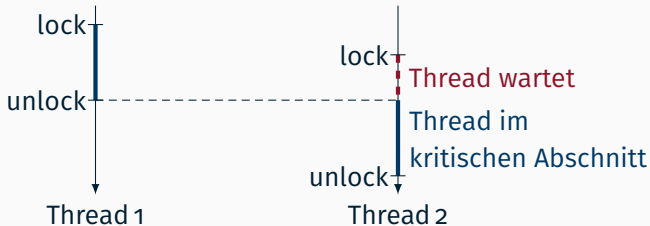
4.4 Statische Bibliotheken

4.5 Dynamische Bibliotheken

4.6 Aufgabe 4: jbuffer



- **Mutual exclusion** (gegenseitiger Ausschluss)
- Koordinierung von kritischen Abschnitten:



- Nur ein Thread kann gleichzeitig den Mutex sperren und somit den kritischen Abschnitt durchlaufen



Schnittstelle

- Mutex erzeugen:

```
pthread_mutex_t m;  
errno = pthread_mutex_init(&m, NULL); // Fehlerbehandlung!
```

- Sperren und freigeben:

```
pthread_mutex_lock(&m);  
// kritischer Abschnitt  
pthread_mutex_unlock(&m);
```

- Mutex zerstören und Ressourcen freigeben:

```
errno = pthread_mutex_destroy(&m); // Fehlerbehandlung!
```

- Alle Pthread-Funktionen setzen **errno** nicht implizit, sondern geben einen Fehlercode zurück (im Erfolgsfall: 0)
- **Randnotiz:** **errno** ist keine globale Variable, sondern eine Thread-lokale Variable – jeder Thread besitzt seine eigene **errno**

- Wie schützen wir die Liste vor Nebenläufigkeit in mehrfädigen Programmen?

```
static volatile QElement *head;

int syncEnqueue(int value) {
    int result = enqueue(value);

    return result;
}
```

```
int syncDequeue(void) {
    while(head == NULL) {
        // Wait
        // for
        // syncEnqueue()
    }

    int value = dequeue();

    return value;
}
```

- Wie schützen wir die Liste vor Nebenläufigkeit in mehrfädigen Programmen?
 - Zugriffe auf enqueue() und dequeue() per Mutex serialisieren
 - Schutz sowohl bei mehreren Einfüge- als auch bei mehreren Entnahme-Threads
- Weiteres Nebenläufigkeitsproblem?

```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);

    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    while(head == NULL) {
        // Wait
        // for
        // syncEnqueue()
    }
    pthread_mutex_lock(&m);
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```

- Problem: Mehrere Entnahme-Threads könnten gleichzeitig in der Schleife warten
 - dequeue() könnte mehrmals aufgerufen werden, obwohl nur ein neues Element eingefügt wurde
 - Lösung?

```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);

    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    while(head == NULL) {
        // Wait
        // for
        // syncEnqueue()
    }
    pthread_mutex_lock(&m);
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```

- Problem: Mehrere Entnahme-Threads könnten gleichzeitig in der Schleife warten
 - dequeue() könnte mehrmals aufgerufen werden, obwohl nur ein neues Element eingefügt wurde
 - Lösung: Warteschleife in den kritischen Abschnitt ziehen
- Problem jetzt vollständig gelöst?

```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);

    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    pthread_mutex_lock(&m);
    while(head == NULL) {
        // Wait
        // for
        // syncEnqueue()
    }
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```

- Problem: Deadlock, da in kritischem Bereich gewartet wird
 - Kein anderer Thread wird den kritischen Abschnitt jemals mehr betreten können
 - Lösung?

```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);

    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    pthread_mutex_lock(&m);
    while(head == NULL) {
        // Wait
        // for
        // syncEnqueue()
    }
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```

- Problem: Deadlock, da in kritischem Bereich gewartet wird
 - Kein anderer Thread wird den kritischen Abschnitt jemals mehr betreten können
 - Lösung: Mutex in der Warteschleife kurzzeitig freigeben
- Um aktives Warten zu vermeiden, ist ein Schlaf/Aufweck-Mechanismus nötig

```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);

    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    pthread_mutex_lock(&m);
    while(head == NULL) {
        pthread_mutex_unlock(&m);
        // Wait for syncEnqueue()
        pthread_mutex_lock(&m);
    }
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```

- Pseudo-Funktionen zur Vermeidung von aktivem Warten:
WAIT_FOR_CHANGE() blockiert so lange, bis SIGNAL_CHANGE() aufgerufen wurde
- Nebenläufigkeitsproblem?: das altbekannte *Lost-Wakeup*-Problem
 - Aufweck-Signalisierung kann verloren gehen
 - Freigabe des Mutex und Schlafenlegen muss atomar erfolgen

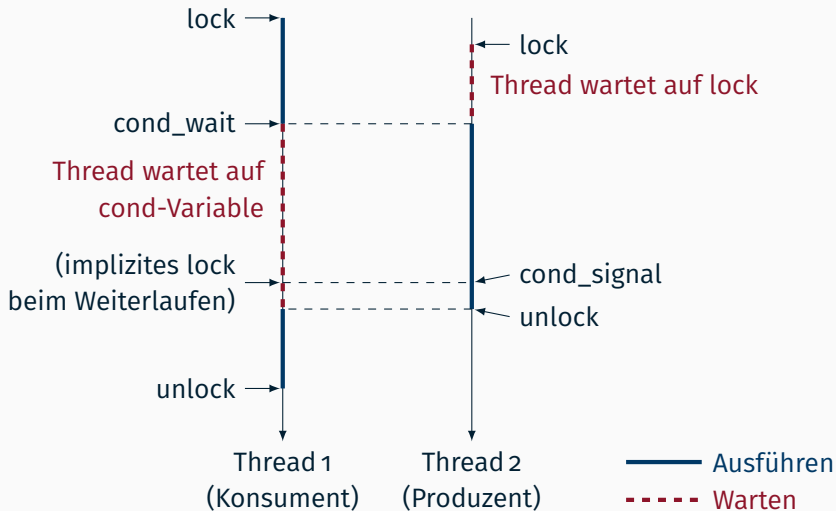
```
static volatile QElement *head;
static pthread_mutex_t m;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);
    SIGNAL_CHANGE();
    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    pthread_mutex_lock(&m);
    while(head == NULL) {
        pthread_mutex_unlock(&m);
        WAIT_FOR_CHANGE();
        pthread_mutex_lock(&m);
    }
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```



- Mechanismus zum Blockieren und Aufwecken von Threads in durch Mutex geschützten kritischen Abschnitten
 - Beinhaltet Warteschlange für das Warten auf ein Ereignis
- `pthread_cond_wait()`:
 - Thread reiht sich in Warteschlange der Bedingungsvariable ein
 - Thread gibt atomar den Mutex frei (*unlock*) und legt sich schlafen
 - Nach Signalisierung wird Thread wieder lafbereit
 - Thread betritt den kritischen Abschnitt neu (*lock*)
- `pthread_cond_signal()` / `pthread_cond_broadcast()`:
 - Aufwecken eines (oder mehrerer) Threads aus der Warteschlange der Bedingungsvariable





- Bei `pthread_cond_signal()` wird **mindestens einer** der wartenden Threads aufgeweckt
- Mit `pthread_cond_broadcast()` werden **alle** wartenden Threads aufgeweckt
 - Der Scheduler entscheidet, welcher Thread als erster weiterläuft
 - Dieser Thread wird als erster den Mutex neu belegen
 - Alle anderen Threads werden dann am Mutex serialisiert
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Schleifenbedingung nach dem Aufwachen nochmals überprüft werden

- Initialisierung von Mutex und Bedingungsvariable mit `pthread_{mutex,cond}_init()`
- Zerstören mit `pthread_{mutex,cond}_destroy()`

```
static volatile QElement *head;
static pthread_mutex_t m;
static pthread_cond_t c;

int syncEnqueue(int value) {
    pthread_mutex_lock(&m);
    int result = enqueue(value);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return result;
}
```

```
int syncDequeue(void) {
    pthread_mutex_lock(&m);
    while(head == NULL) {
        pthread_cond_wait(&c, &m);
    }
    int value = dequeue();
    pthread_mutex_unlock(&m);
    return value;
}
```



4.1 Mutexe und Bedingungsvariablen

4.2 Nichtblockierende Synchronisation

4.3 Module und Symbole

4.4 Statische Bibliotheken

4.5 Dynamische Bibliotheken

4.6 Aufgabe 4: jbuffer



- Nichtblockierende Synchronisation wird üblicherweise mit Hilfe der *Compare-and-swap-Operation* (CAS) implementiert (→ siehe Vorlesung C | X.4, Seite 20 ff.)
- Funktionsweise von CAS:
 - Argumente: Speicheradresse, erwarteter Wert, neuer Wert
 - Atomare Operation:
 - Falls in der Speicherstelle der erwartete Wert steht, überschreibe sie mit dem neuen Wert und gib `true` zurück
 - Andernfalls lasse die Speicherstelle unverändert und gib `false` zurück
- Verwendung:

```
do {  
    // Ziehe lokale Kopie der kritischen Variable  
    // Berechne lokal neuen Wert  
} while(CAS(/* krit. Variable, Wert alt, Wert neu */) == false);
```

- Falls die kritische Variable nebenläufig verändert wurde, wird der kritische Abschnitt wiederholt
- Achtung: genau überlegen, wie der kritische Abschnitt aussehen muss!



Die CAS-Operation selbst lässt sich nicht atomar in C implementieren

- Möglichkeit 1: Inline-Assembly
- Möglichkeit 2: GCC-*Builtin*-Funktion

```
bool __sync_bool_compare_and_swap(type *ptr, type oldval,  
                                  type newval);
```

- Möglichkeit 3: `<stdatomic.h>` (ab Sprachstandard C11)
 - (Leider nur optionales Feature)
 - *Unterstützt in GCC seit Version 4.9*

```
bool atomic_compare_exchange_strong(volatile A* obj,  
                                    C* expected, C desired);
```



```
#include <stdatomic.h>
```

atomic-Datentypen in C11

Schlüsselwort `_Atomic`: Kennzeichnung von **atomic**-Variablen.

Zusätzlich vorgegebene typedefs: `atomic_int`, `atomic_long`...

- Initialisieren: `atomic_init(volatile A *object, C value)`

```
_Atomic int x;  
atomic_init(&x, 42);
```

- Lesen: `C atomic_load(volatile A *object)`

```
int y = atomic_load(&x);
```

- Schreiben:

```
void atomic_store(volatile A *object, C desired)
```

```
atomic_store(&x, 47);
```



```
#include <stdatomic.h>
bool atomic_compare_exchange_strong(
    volatile A* obj, C* expected, C desired);
```

- `obj`: Zeiger auf zu modifizierendes **atomic**-Datum
- `expected`: Zeiger auf den erwarteten Wert
- `desired`: Der neue Wert

Doku & Details unter <https://en.cppreference.com/w/c/atomic>



4.1 Mutexe und Bedingungsvariablen

4.2 Nichtblockierende Synchronisation

4.3 Module und Symbole

4.4 Statische Bibliotheken

4.5 Dynamische Bibliotheken

4.6 Aufgabe 4: jbuffer



```
#include <stdio.h>
#include "bar.h"

int main(void) {
    bar(42);
}
```

main.c

```
#ifndef BAR_H
#define BAR_H

    void bar(int);

#endif
```

bar.h (Schnittstelle)

- Module exportieren eine Schnittstelle (Header-Datei):
 - Funktionsdeklarationen
 - Gegebenenfalls Deklarationen (`extern`) globaler Variablen
 - *Include-Guard* (`#ifndef`-Konstrukt) verhindert Mehrfachdeklaration (von Variablen, Structs, etc.), falls der Header mehrfach eingebunden wird
- Beim Übersetzen muss Compiler den Typ eines Symbols kennen:
 - Einbinden der Schnittstellenbeschreibung mit `#include "bar.h"`
 - gcc-Parameter `-Ipfad`: teilt Compiler zusätzlichen Suchpfad für Header-Dateien mit (aktuelles Verzeichnis ist immer enthalten)



- Der Zugriff auf Funktionen und globale Variablen erfolgt in C-Programmen über **symbolische Namen**
- Der Namensraum ist flach und nicht typisiert:
 - Jeder Name muss eindeutig sein
 - Es darf z. B. keine Funktion mit dem Namen einer globalen Variable geben
- Kompilierte Übersetzungseinheit (.o-Datei) enthält **Symboltabelle**:
 - Liste von Symbolen, die von der Einheit **definiert** werden
 - Liste von Symbolen, die von der Einheit **verwendet** werden

Anzeige von Symboltabellen mit dem Programm nm(1)

- Für definierte Symbole: Anzeige des Offsets im Segment (im gebundenen Programm stattdessen absolute Adresse)
- Segment: U = unresolved, B = .bss, D = .data, T = .text
 - Sichtbarkeit: groß = globales Symbol, klein = modullokales Symbol



```
#include "bar.h"
int main(void) {
    bar(42);
}
```

```
U bar
00000000 T main
```

Modul main

```
#ifndef BAR_H
#define BAR_H
void bar(int);
#endif
```

bar.h (Schnittstelle)

```
#include "bar.h"
void bar(int param) {
    // Do stuff
}
```

```
00000000 T bar
```

Modul bar

- Modul bar *definiert* Symbol bar (Funktion `void bar(int)`)
- Hauptprogramm *verwendet* Symbol bar (ruft die Funktion `void bar(int)` auf)



- Linker bindet die angegebenen Übersetzungseinheiten zu einem ausführbaren Binärabbild im ELF-Format zusammen
- Offene Symbolreferenzen werden aufgelöst:
 1. Suche in anderen Übersetzungseinheiten
 2. Suche in der Standard-C-Bibliothek (libc)
- Fehler, falls nicht alle offenen Symbolreferenzen aufgelöst werden können (*undefined reference*)
- Fehler, falls ein Symbol mehrfach definiert ist (*duplicate symbol*)



4.1 Mutexe und Bedingungsvariablen

4.2 Nichtblockierende Synchronisation

4.3 Module und Symbole

4.4 Statische Bibliotheken

4.5 Dynamische Bibliotheken

4.6 Aufgabe 4: jbuffer



- Statische Bibliothek:
 - (Unkomprimiertes) Archiv, in dem mehrere Objekt-Dateien (.o) zusammengefasst sind
 - Enthält eigene Symboltabelle
 - Übliche Dateinamenskvention: `libexample.a`
- Erstellen mit dem Kommando `ar(1p)`:
`user@host:~$ ar -rcs libexample.a bar.o foo.o`



- Bibliothek kann dem Linker als Symbolquelle angeboten werden:
 - Parameter `-lname`: Binden mit der Bibliothek `libname.a`
 - Diese Datei wird in einer Menge von Suchpfaden gesucht
 - Voreingestellte System-Suchpfade: z. B. `/usr/local/lib`, `/usr/lib`, ...
 - Parameter `-Lpath`: `path` als Suchpfad hinzufügen
 - Betrifft nur nachfolgende Vorkommen von `-l`
- Linker bindet dann alle `.o`-Dateien aus der Bibliothek, die **bis dahin** unaufgelöste Symbole definieren, zum Binärabbild dazu
 - Relative Reihenfolge von Objekt-Dateien und Bibliotheken ist wichtig – Bibliotheken sollten i. d. R. **am Schluss** angegeben werden
- Bibliothek wird zur Ausführung des Programms nicht mehr benötigt



- 4.1 Mutexe und Bedingungsvariablen
- 4.2 Nichtblockierende Synchronisation
- 4.3 Module und Symbole
- 4.4 Statische Bibliotheken
- 4.5 Dynamische Bibliotheken**
- 4.6 Aufgabe 4: jbuffer



- Dynamische Bibliothek (*Shared Library*):
 - Kein Dateiarchiv, sondern eine ladbare Funktionssammlung
 - Bibliothek wird zur Ausführung des Programms benötigt
 - Übliche Namenskonvention: `libexample.so`
- Code liegt nach dem Laden i. d. R. nur einmal im Hauptspeicher, kann aber in verschiedenen Prozessen an unterschiedlichen Adressen im logischen Adressraum positioniert sein
 - Keine absoluten Adressen (Funktionsaufrufe, globale Variablen) im Maschinencode der Bibliothek erlaubt
 - PIC (*Position-Independent Code*, gcc-Option `-fPIC`)
- Bibliotheksmodule mit `-fPIC` kompilieren
- Bibliothek durch Zusammenbinden der `.o`-Dateien erstellen:

```
user@host:~$
```

```
gcc -shared $(LDFLAGS) $(CFLAGS) -o libexample.so bar.o foo.o
```



- Binden einer dynamischen Bibliothek an eine Anwendung:
 - Linker-Aufruf identisch zu statischem Binden (Flags `-l` und `-L`)
 - Aber kein Kopieren der `.o`-Dateien, sondern nur Anlegen von Verweisen im ELF-Binary
 - Falls in den Suchpfaden sowohl eine statische als auch eine dynamische Bibliothek gefunden wird, wird die dynamische gewählt
 - Relative Reihenfolge von Objekt- bzw. Quelldateien und Bibliotheken ist u. U. ebenfalls wichtig
- Das endgültige Binden erfolgt erst beim Laden:
 - Beim Laden des Programms (`exec(3p)`) wird zunächst der *Dynamic Linker/Loader* (`ld.so`) geladen
 - `ld.so` lädt das Programm und die Bibliothek (sofern noch nicht im Hauptspeicher vorhanden) und bindet noch offene Referenzen
 - Bibliothek wird von `ld.so` in mehreren Verzeichnissen gesucht (über Umgebungsvariable `LD_LIBRARY_PATH` einstellbar)



- Hauptvorteile von dynamischen Bibliotheken:
 - Insgesamt geringerer Platten- und Hauptspeicherverbrauch
 - Üblicherweise zentraler Installationsort (z. B. `/usr/lib`):
 - Bei einem Update (u. U. sicherheitskritisch!) muss nur eine Datei ausgetauscht werden
 - Kein erneutes Binden aller betroffener Anwendungen nötig
- Vollständig statisches Binden ist auf PCs kaum mehr gebräuchlich:
 - `libc` wird fast immer dynamisch gebunden, Bibliotheken zumeist abhängig von Programmiersprache
 - Manche Betriebssysteme (z. B. macOS, Solaris 10) bieten gar keine statische `libc` mehr



- 4.1 Mutexe und Bedingungsvariablen
- 4.2 Nichtblockierende Synchronisation
- 4.3 Module und Symbole
- 4.4 Statische Bibliotheken
- 4.5 Dynamische Bibliotheken
- 4.6 Aufgabe 4: jbuffer**



Ringpuffer-Modul

- Ringpuffer zur Verwaltung von `int`-Werten
 - Zutatenliste: Array, Leseindex, Schreibindex, Modulo-Operation
- Randbedingung: ein Produzent, mehrere Konsumenten
- Blockierende Synchronisation zwischen Produzenten und Konsumenten mittels Semaphoren zur Vermeidung von Über- bzw. Unterlauf
- Nichtblockierende Synchronisation der Konsumenten untereinander mittels CAS (siehe Vorlesung C | X.4, Seite 20 ff.)

Semaphor-Modul

- Zählender P/V-Semaphor (siehe Vorlesung C | X.3, Seite 6 ff.)