# Übungen zu Systemprogrammierung 2 (SP2)

## Ü4 – Thread-Koordinierung

**Christoph Erhardt, Jens Schedel, Jürgen Kleinöder**

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 10. bis 14. Juni 2013

http://www4.cs.fau.de/Lehre/SS13/V_SP2



## Agenda



24-ThreadSync_handout



## Agenda

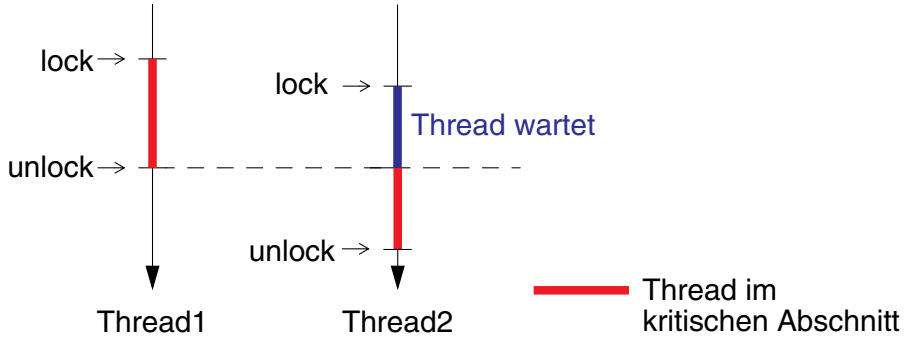


24-ThreadSync_handout



## Mutexe

- Koordinierung von kritischen Abschnitten:




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

24-ThreadSync_handout

## Pthread-Mutexe

- Schnittstelle:
  - Mutex erzeugen:

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

  - Sperren und freigeben:

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

  - Mutex zerstören und Ressourcen freigeben:

```
errno = pthread_mutex_destroy(&m);
```

- Alle Pthread-Funktionen setzen `errno` nicht implizit, sondern geben einen Fehlercode zurück (im Erfolgsfall: 0)



## Beispiel: Semaphor-Implementierung



- Welches Problem kann hier auftreten?

```
static volatile int a;

void P(void) {
    while (a == 0) {
        // Wait for change
    }
    --a;
}
```

```
void V(void) {
    ++a;
}
```



## Beispiel: Semaphor-Implementierung



- Welches Problem kann hier auftreten?
  - *Lost-Update*-Problem, da Inkrement und Dekrement nicht atomar
  - Lösung: Zugriff auf `a` mit Mutex schützen
- Weiteres Nebenläufigkeitsproblem?

```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    while (a == 0) {
        // Wait for change
    }
    pthread_mutex_lock(&m);
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_mutex_unlock(&m);
}
```



## Beispiel: Semaphor-Implementierung



- Problem: Mehrere Threads könnten gleichzeitig in Schleife warten
  - `a` könnte mehrmals heruntergezählt werden
  - Lösung?

```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    while (a == 0) {
        // Wait for change
    }
    pthread_mutex_lock(&m);
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_mutex_unlock(&m);
}
```

## Beispiel: Semaphor-Implementierung



- Problem: Mehrere Threads könnten gleichzeitig in Schleife warten
  - a könnte mehrmals heruntergezählt werden
  - Lösung: Prüfung der Bedingung in den kritischen Abschnitt ziehen
- Problem jetzt vollständig gelöst?

```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    pthread_mutex_lock(&m);
    while (a == 0) {
        // Wait for change
    }
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_mutex_unlock(&m);
}
```



## Beispiel: Semaphor-Implementierung



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

```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    pthread_mutex_lock(&m);
    while (a == 0) {
        // Wait for change
    }
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_mutex_unlock(&m);
}
```



## Beispiel: Semaphor-Implementierung



- Problem: Deadlock, da in kritischem Bereich gewartet wird
  - Kein anderer Thread kann den kritischen Abschnitt betreten
  - Lösung: Mutex während des Wartens freigeben
- Aktives Warten vermeiden: Schlaf/Aufweck-Mechanismus nötig

```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    pthread_mutex_lock(&m);
    while (a == 0) {
        pthread_mutex_unlock(&m);
        // Wait for change
        pthread_mutex_lock(&m);
    }
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_mutex_unlock(&m);
}
```



## Passives Warten auf ein Ereignis



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

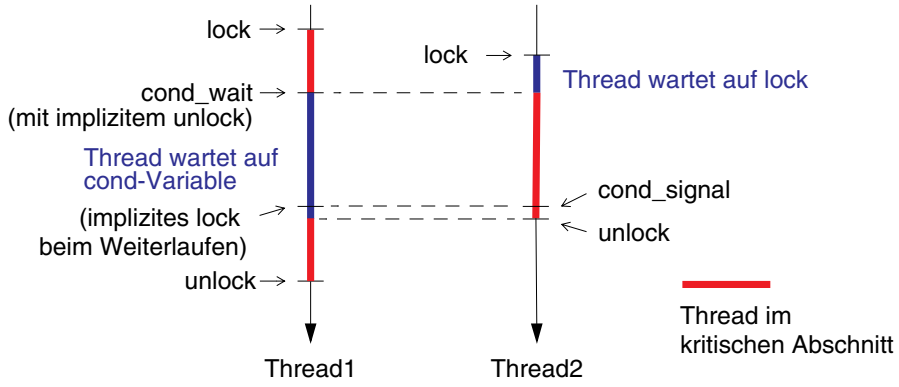
```
static volatile int a;
static pthread_mutex_t m;

void P(void) {
    pthread_mutex_lock(&m);
    while (a == 0) {
        pthread_mutex_unlock(&m);
        WAIT_FOR_CHANGE();
        pthread_mutex_lock(&m);
    }
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    SIGNAL_CHANGE();
    pthread_mutex_unlock(&m);
}
```

# Pthread-Bedingungsvariablen

- Mechanismus zum Blockieren und Aufwecken von Threads:




- Freigeben des aktuellen kritischen Abschnitts beim Blockieren
- Betreten des kritischen Abschnitts nach dem Aufwachen



# Beispiel: Semaphor-Implementierung



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

```
static pthread_mutex_t m;
static pthread_cond_t  c;
static volatile int a;

void P(void) {
    pthread_mutex_lock(&m);
    while (a == 0) {
        pthread_cond_wait(&c, &m);
    }
    --a;
    pthread_mutex_unlock(&m);
}
```

```
void V(void) {
    pthread_mutex_lock(&m);
    ++a;
    pthread_cond_broadcast(&c);
    pthread_mutex_unlock(&m);
}
```



# Pthread-Bedingungsvariablen

- Realisierung von `pthread_cond_wait()`:
  - Thread reiht sich in Warteschlange der Bedingungsvariable ein
  - Thread gibt Mutex frei
  - Thread gibt Prozessor auf
  - Nach Signalisierung wird Thread wieder laufbereit
  - Thread muss kritischen Abschnitt neu betreten (*lock*)
- Realisierung von `pthread_cond_broadcast()` / `pthread_cond_signal()`:
  - Aufwecken eines (oder mehrerer) Threads aus der Warteschlange der Bedingungsvariablen
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden



# Pthread-Bedingungsvariablen

- Bei `pthread_cond_signal()` wird mindestens einer der wartenden Threads aufgeweckt – es ist allerdings u. U. nicht definiert, welcher
  - Eventuell Prioritätsverletzung, wenn nicht der höchstpriore gewählt wird
  - Verklemmungsgefahr, wenn die Threads unterschiedliche Wartebedingungen haben
- Mit `pthread_cond_broadcast()` werden alle wartenden Threads aufgeweckt
- Ein aufwachender Thread wird als erstes den Mutex neu belegen – ist dieser gerade gesperrt, bleibt der Thread solange blockiert

# Agenda







# Nichtblockierende Synchronisation

- Nichtblockierende Synchronisation wird üblicherweise mit Hilfe der *Compare-and-swap*-Operation (*CAS*) implementiert (→ siehe Vorlesung C | X-4, Seite 20f.)
- Die CAS-Operation selbst lässt sich nicht atomar in C implementieren
  - Möglichkeit 1: Inline-Assembly
    - In den C-Code eingebettete Sequenz von Maschineninstruktionen
    - Schlechte Portierbarkeit: Syntax ist Compiler- und CPU-spezifisch
  - Möglichkeit 2: Compiler-*Builtin*-Funktion
    - Verwendung wie eine gewöhnliche Funktion
    - Statt eines Funktionsaufrufs erzeugt der Compiler eine Sequenz von Maschineninstruktionen für den jeweiligen Zielprozessor

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





# Agenda







# Compiler und Module

```c
#include "bar.h"

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

`main.c`

```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
- Beim Übersetzen muss Compiler den Typ eines Symbols kennen:
  - Einbinden der Schnittstellenbeschreibung mit `#include "bar.h"`
    - `gcc`-Parameter `-I`*`pfad`*: teilt Compiler zusätzlichen Suchpfad für Header-Dateien mit (aktuelles Verzeichnis ist immer enthalten)



# Symbole

- 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 verwendet werden
  - Liste von Symbolen, die von der Einheit definiert werden

## Anzeige von Symboltabellen mit dem Programm `nm(1)`

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



# Linker und Module

```c
#include "bar.h"

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

```
         U bar
00000000 T main
```

Modul `main`

```c
#ifndef BAR_H
#define BAR_H

void bar(int);

#endif
```

`bar.h` (Schnittstelle)

```c
#include "bar.h"

void bar(int param) {
    // Do stuff
}
```

```
00000000 T bar
```

Modul `bar`

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



# Agenda





# Statische Bibliotheken

- Statische Bibliothek:
  - (Unkomprimiertes) Archiv, in dem mehrere Objekt-Dateien (`.o`) zusammengefasst sind
  - Enthält eigene Symboltabelle
  - Übliche Dateinamenskonvention: `libexample.a`
- Erstellen mit dem Kommando `ar(1)`:

```
ar -rcs libexample.a bar.o foo.o
```

- Bibliothek kann dem Linker als Symbolquelle angeboten werden
- Bibliothek wird zur Ausführung des Programms nicht mehr benötigt

# Binden von Programmen

- Statisches Binden: Zusammenbinden der angegebenen Übersetzungseinheiten zu einem ausführbaren Binärabbild
  - Offene Symbolreferenzen werden aufgelöst
    - Definiert in anderen Übersetzungseinheiten
    - Suche in Programmbibliotheken
  - GCC sucht beim Binden implizit in der Standard-C-Bibliothek (`libc.a`)
- Weitere Bibliotheken können vom Entwickler angegeben werden
  - Parameter `-L`*`path`*: Suche nach Bibliotheken (`.a`-Dateien) in *`path`*
    - Standard-Suchpfade: `/usr/local/lib`, `/usr/lib`
  - Parameter `-l`*`name`*: Binden mit der Bibliothek `lib`*`name`*`.a`
    - Diese Datei wird in den Suchpfaden gesucht
  - Linker bindet dann alle `.o`-Dateien aus der Bibliothek, die **bis dahin** unaufgelöste Symbole definieren, zum Binärabbild dazu
  - Die Reihenfolge von Objekt-Dateien und Bibliotheken ist wichtig!





# Agenda









# Dynamische Bibliotheken

- Dynamische Bibliothek (*Shared Library*):
  - Kein Dateiarchiv, sondern eine ladbare Funktionssammlung
  - Bibliothek wird zur Ausführung des Programms benötigt
  - Übliche Dateinamenskonvention: `lib`*`example`*`.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 erlaubt
    → PIC (*Position-Independent Code*, `gcc`-Option `-fPIC`)
- Erstellen der Bibliothek durch Zusammenbinden der `.o`-Dateien:

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







# Dynamische Bibliotheken

- 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 Binary
  - Falls in den Suchpfaden sowohl eine statische als auch eine dynamische Bibliothek gefunden wird, wird die dynamische gewählt
  - GCC sucht beim Binden implizit in der Standard-C-Bibliothek (`libc.so`)
  - Reihenfolge von Bibliotheken und Objekt- bzw. Quelldateien ist u. U. ebenfalls wichtig
- Das endgültige Binden erfolgt erst beim Laden:
  - Beim Laden des Programms (`exec(2)`) 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)



# Verwendung von dynamischen Bibliotheken

- 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 und andere Bibliotheken werden fast immer dynamisch gebunden
  - Manche Betriebssysteme (z. B. OS X, Solaris 10) bieten gar keine statische libc mehr



# Agenda





# Aufgabe 4: `jbuffer`

## Ringpuffer-Modul

- Ringpuffer zur Verwaltung von `int`-Werten
- 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 20f.)

## Semaphor-Modul

- Zählender P/V-Semaphor zur Synchronisation von POSIX-Threads (siehe Vorlesung C | X-3, Seite 19f.)