U5 POSIX-Threads

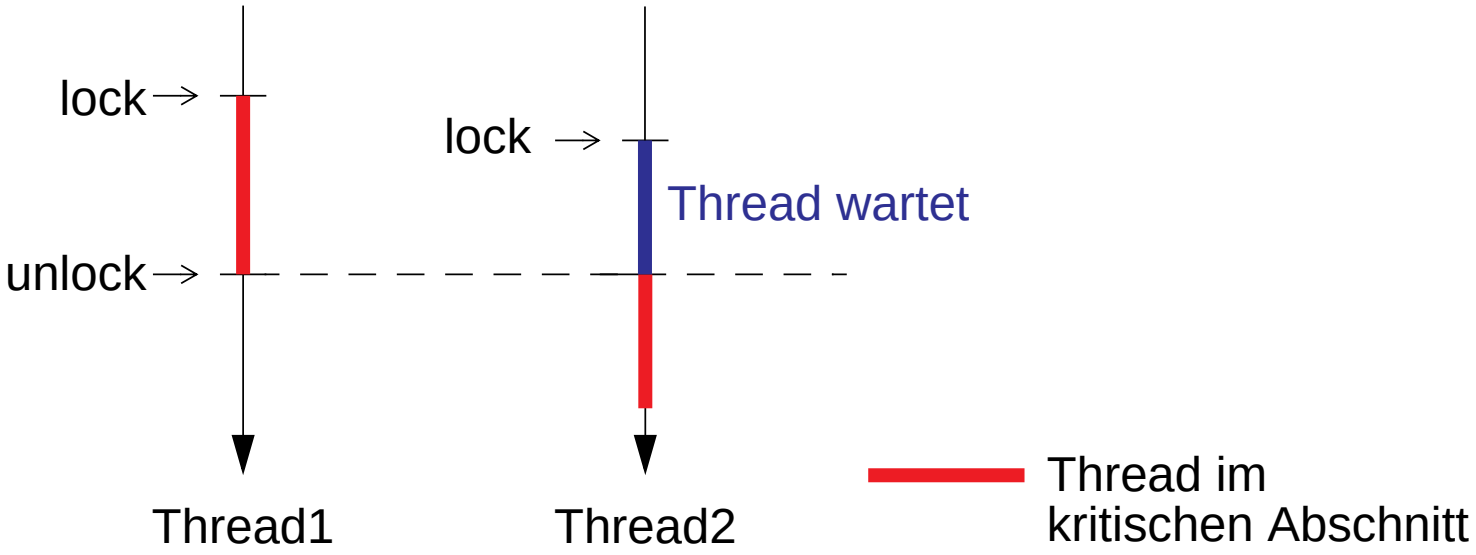
- Hinweise zur Evaluation
- Mutexe
- Sychronisierung mit Mutexen
- Bedingungsvariablen (Condition Variables)
- Nichtblockierende Synchronisation
- Statische und dynamische Bibliotheken
- Aufgabe 4: *jbuffer*

U5-1 Hinweise zur Evaluation

- in Kommentaren, die sich auf einen bestimmten Übungsleiter beziehen, bitte dessen Namen **jeder Zeile** voranstellen
- Frage "eigener Aufwand zur Vor- und Nachbereitung"
  - bitte nach Vorlesung und Übung auftrennen
  - Übung: den jeweiligen Wochenaufwand durch 2 teilen (rechnerisch: 2x1 Stunde Übung (Tafel+Rechner) pro Woche, Angabe je 45 Minuten)
  - Vorlesung: den jeweiligen Wochenaufwand nicht teilen (1x90 Minuten Vorlesung pro Woche, Aufwandsangabe je 90 Minuten)
- Vorlesungsevaluation: "Dozent hat Vorlesung zu ... selbst gehalten"
  - Dozenten sind Prof. Schröder-Preikschat und Dr. Jürgen Kleinöder
  - technisch bedingt wird in der Evaluation nur Dr. Jürgen Kleinöder als Dozent genannt
- bitte beide Dozenten bei der Beantwortung der Frage berücksichtigen

U5-2 Mutexe

- Koordinierung von kritischen Abschnitten



U5-2 Mutexe

- Schnittstelle:
  - Mutex erzeugen

```
pthread_mutex_t m1;
errno = pthread_mutex_init(&m1, NULL);
```

  - Lock & unlock

```
errno = pthread_mutex_lock(&m1);
... kritischer Abschnitt
errno = pthread_mutex_unlock(&m1);
```

  - Mutex zerstören und Ressourcen freigeben

```
errno = pthread_mutex_destroy(&m1);
```

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

U5-3 Synchronisierung mit Mutexen

- ... am Beispiel einer Semaphor-Implementierung
- Welches Problem kann hier auftreten?

```
static pthread_mutex_t m1;
static volatile int a;

void P(void) {
    while (a == 0) {
        // wait for changes
    }
    pthread_mutex_lock(&m1);
    --a;
    pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
    pthread_mutex_lock(&m1);
    ++a;
    pthread_mutex_unlock(&m1);
}
```

U5-3 Synchronisierung mit Mutexen

- Problem: Mehrere Threads warten gleichzeitig in der Schleife
  - `a` wird mehrmals heruntergezählt
- Prüfung der Bedingung ist ebenfalls Teil des kritschen Abschnittes
- Welches Problem ist nun vorhanden?

```
static pthread_mutex_t m1;
static volatile int a;

void P(void) {
    pthread_mutex_lock(&m1);
    while (a == 0) {
        // wait for change
    }
    --a;
    pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
    pthread_mutex_lock(&m1);
    ++a;
    pthread_mutex_unlock(&m1);
}
```

U5-3 Synchronisierung mit Mutexen

- Deadlock, da in kritischem Bereich gewartet wird
  - Kein anderer Thread kann den kritischen Abschnitt betreten
  - Freigabe des Mutexes während des Wartens notwendig

```
static pthread_mutex_t m1;
static volatile int a;

void P(void) {
    pthread_mutex_lock(&m1);
    while (a == 0) {
        pthread_mutex_unlock(&m1);
        // wait for change
        pthread_mutex_lock(&m1);
    }
    --a;
    pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
    pthread_mutex_lock(&m1);
    ++a;
    pthread_mutex_unlock(&m1);
}
```

- Zur Vermeidung von aktivem Warten wird ein Sleep/Wakeup-Mechanismus benötigt.

U5-4 Synchronisierung mit Mutexen und Sleep/ Wakeup-Mechanismus

- Zur Vermeidung von aktivem Warten wird Pseudo-Funktion `wait_for_change()` eingesetzt, diese blockiert solange bis `signal_change()` aufgerufen wird.

```
static pthread_mutex_t m1;
static volatile int a;

void P(void) {
    pthread_mutex_lock(&m1);
    while (a == 0) {
        pthread_mutex_unlock(&m1);
        wait_for_change();
        pthread_mutex_lock(&m1);
    }
    --a;
    pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
    pthread_mutex_lock(&m1);
    ++a;
    pthread_mutex_unlock(&m1);
    signal_change();
}
```

- Welches Problem kann hier auftreten?

U5-4 Synchronisierung mit Mutexen und Sleep/ Wakeup-Mechanismus

- Lost-Wakeup-Problem
  - Aufwecksignal kann verloren gehen
  - Freigabe des Mutex und Schlafenlegen muss atomar sein!

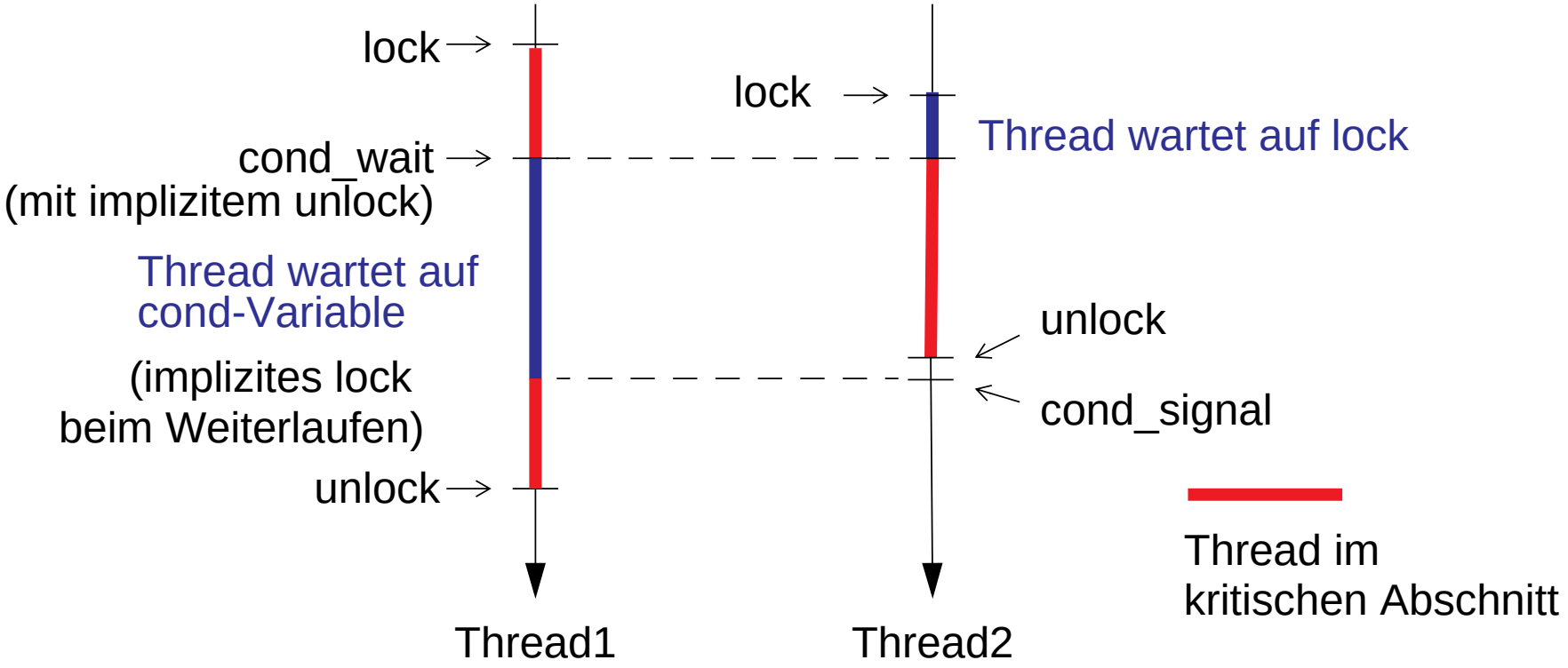
```
static pthread_mutex_t m1;
static volatile int a;

void P(void) {
    pthread_mutex_lock(&m1);
    while (a == 0) {
        // atomic start
        pthread_mutex_unlock(&m1);
        wait_for_change();
        pthread_mutex_lock(&m1);
        // atomic end
    }
    --a;
    pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
    pthread_mutex_lock(&m1);
    ++a;
    pthread_mutex_unlock(&m1);
    signal_change();
}
```

U5-5 Bedingungsvariablen - Condition-Variables

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



U5-5 Bedingungsvariablen

- ... am Beispiel der Semaphor-Implementierung

```
static pthread_cond_t c1;
static pthread_mutex_t m1;
static volatile int a;

int main(void) {
   pthread_cond_init(&c1, NULL);
   // ...
}

void P(void) {
   pthread_mutex_lock(&m1);
   while (a == 0) {
      pthread_cond_wait(&c1, &m1);
   }
   a--;
   pthread_mutex_unlock(&m1);
}
```

```
void V(void) {
   pthread_mutex_lock(&m1);
   a++;
   pthread_cond_broadcast(&c1);
   pthread_mutex_unlock(&m1);
}
```

U5-5 Bedingungsvariablen

- Realisierung `cond_wait()`
  - Thread reiht sich in Warteschlange der Bedingungsvariablen ein
  - Thread gibt Mutex frei
  - Thread gibt Prozessor auf
  - Nach Signalisierung wird Thread wieder laufbereit (`cond_signal()` s.u.)
  - Thread muss kritischen Abschnitt neu betreten (`lock()`)
- Realisierung `cond_broadcast()`/`cond_signal()`
  - Aufwecken eines (oder mehrerer) Threads, aus der Warteschlange der Bedingungsvariablen
- Da möglicherweise mehrere Threads deblockiert wurden, muss die Bedingung nochmals überprüft werden (siehe Seite 8)

U5-5 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

U5-6 Nichtblockierende Synchronisation

- Zur Erhaltung von Portabilität verwenden wir sogenannte *GCC-Builtins*
  - Verwendung wie eine gewöhnliche Funktion
  - Statt eines Funktionsaufrufs erzeugt der Compiler aber eine Sequenz von Maschineninstruktionen für den Zielprozessor
- Zur nichtblockierenden Sychronisation mittels CAS verwenden wir folgendes *GCC-Builtin*:

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

- Zur Verwendung von CAS siehe Vorlesung C X-4 Seite 20 f.

U5-7 Überblick C-Funktionsbibliotheken

- Statische Bibliothek:
  - ➤ Archiv, in dem mehrere Objekt-Dateien (.o) zusammengefasst werden
  - ➤ beim statischen Binden eines Programms werden die benötigten Objekt-Dateien zu der ausführbaren Datei hinzukopiert
  - ➤ Bibliothek ist bei der Ausführung des Programms nicht mehr sichtbar
- Dynamische, gemeinsam genutzte Bibliothek (*shared library*):
  - ➤ Zusammenfassung von übersetzten C-Funktionen
  - ➤ beim Binden werden Referenzen auf die Funktionen offen gelassen
  - ➤ Shared Library ist nur einmal im Hauptspeicher vorhanden
  - ➤ Shared Library wird in virtuellen Adressraum dynamisch gebundener Programme beim Laden eingeblendet,
    noch offene Referenzen werden danach gebunden

U5-7 Symboltabellen

- Zugriff auf Funktionen und globale Variablen über symbolische Namen
- Eine Übersetzungseinheit (C-Datei) **definiert** und **verwendet** Symbole
  - Hauptprogramm definiert das Symbol *main* für die main-Funktion
  - Programm ruft (verwendet) eine Funktion der C-Bibliothek (z. B. malloc)
  - C-Datei definiert eine globale (nicht-static) Variable
  - Programm verwendet eine globale Variable, die u. U. in einer anderen Übersetzungseinheit definiert wurden
- Der Namensraum ist flach
  - jeder Name muss eindeutig sein
  - es darf auch keine Funktion mit dem Namen einer globalen Variable geben
- Kompilierte Übersetzungseinheit (Objekt-Datei) enthält Symboltabelle
  - Liste mit Symbolen, die von der Einheit verwendet werden
  - Liste von Symbolen, die von der Einheit definiert werden

U5-7 Symboltabellen

- Anzeige von Symboltabellen mit dem Programm **nm(1)**

```
erhardt@faui0sr0[aufgabe3] nm plist.o
         U free
00000000 b head
         U malloc
000000e0 T plistAdd
00000000 T plistIterate
00000040 T plistRemove
         U strdup
         U strlen
         U strncpy
```

- Offsets im Segment für definierte Symbole (ungebundene Objektdatei)
- Segment: U = unresolved, B = .bss, D = .data, T = .text
  - ◆ Sichtbarkeit: groß = globales Symbol, klein = modullokales Symbol

U5-8 Statische Bibliotheken

- Statisches Binden: Binden der Übersetzungseinheiten zum 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

1 Statische Bibliotheken erstellen

- **Archiv** (Dateiendung **.a**) von Objekt-Dateien (**.o**-Dateien)
  ☞ Erstellen einer Bibliothek mit dem Kommando **ar**

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

- Jede Objekt-Datei enthält wiederum eine Symboltabelle
  ☞ Anzeige mit dem Kommando **nm libexample.a** bzw. **nm bar.o**

- Die Bibliothek kann dem Linker als Symbolquelle angeboten werden
  - Parameter -L*pfad*: Suche nach Bibliotheken (.a-Dateien) in *pfad*
  - Parameter -l*libname*: Binden mit der Bibliothek *libname*
    ☞ diese wird in einer Datei lib*libname*.a in den Suchpfaden gesucht

- Der Linker bindet dann alle Objekt-Dateien, die **bis dahin** unaufgelöste Symbole definieren, zum Binärabbild dazu
  - Die Reihenfolge von Objekt-Dateien und Bibliotheken ist wichtig

2 Beispiel: Kompilieren mit statischen Bibliotheken

```
#include "bar.h"

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

main.c

```
#ifndef BAR_H
#define BAR_H

void bar(int);

#endif
```

bar.h (Schnittstelle)

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

bar.c (Implementierung)

- Module exportieren eine Schnittstelle (Header-Datei)
  - Funktionsdeklarationen und ggf. Deklarationen (*extern*) globaler Variablen
- Beim Übersetzen muss der Compiler den Typen eines Symbols kennen
  - Einbinden der Schnittstellenbeschreibung mit **`#include "bar.h"`**
  - Die Adresse der Funktion bzw. Variable ist an dieser Stelle nicht notwendig
- Parameter -I*pfad*: Teilt Compiler zusätzlichen Suchpfad für Headerdateien mit (das aktuelle Verzeichnis ist immer enthalten)
  - **`gcc -c <...> -Imyincludes main.c`**

3 Beispiel: Binden mit statischen Bibliotheken

```
#include "bar.h"

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

main.c

```
#ifndef BAR_H
#define BAR_H

void bar(int);

#endif
```

bar.h (Schnittstelle)

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

bar.c (Implementierung)

- Modul *bar* **definiert** Symbol *bar* (Funktion `void bar(int);`)
- Hauptprogramm ruft die Funktion `bar` auf (**verwendet** Symbol *bar*)
- Annahme: Modul *bar.o* ist Teil der Bibliothek *libexample.a* in *libdir*
- Übersetzung mit Kommando:

```
gcc -Llibdir -o main <...> main.o -lexample
```

- Falsch (warum?):
  - **`gcc -Llibbdir -o main <...> -lexample main.o`**

U5-9 Shared Libraries

- Kein Dateiarchiv, sondern eine ladbare Funktionssammlung
  - ➤ Erzeugen mit GCC
- Code der Funktionen liegt nur einmal im Hauptspeicher, kann aber in verschiedenen Anwendungen an unterschiedlichen Adressen im logischen Adressraum positioniert sein
  - ➤ keine absoluten Adressen (Sprünge, Unterprogrammaufrufe) im Code erlaubt -> PIC (*position-independent code*)
  - ➤ muss beim Kompilieren der Quellen berücksichtigt werden

```
gcc -fPIC -c file1.c
gcc -fPIC -c file2.c
...
```

◆ Bibliothek wird durch Binden mehrerer .o-Dateien erzeugt

```
gcc -shared -o libutil.so file1.o file2.o ...
```

1 Shared Libraries

- Beim Binden einer Anwendung werden Funktionen nicht aus der Bibliothek kopiert

```
gcc prog.c -L. -lutil -o prog
```

  - ➤ Aufruf analog zum statischen Binden
  - ➤ Bibliothek `lib`*`util`*`.so` wird gesucht

- Das endgültige Binden erfolgt erst beim Laden
  - ➤ Beim Laden von prog (exec) wird zunächst der *dynamic linker/loader* (`ld.so`) geladen
  - ➤ ld.so lädt prog und die Bibliothek (wenn noch nicht im Hauptspeicher vorhanden) und bindet noch offene Referenzen
  - ➤ Bibliothek wird von `ld.so` in mehreren Verzeichnissen gesucht (über Umgebungs-Variable `LD_LIBRARY_PATH` einstellbar)

- Reihenfolge von Bibliotheken und Objekt- bzw. Quelldateien unwichtig
  - ◆ Mehrfachangabe einer Bibliothek nicht notwendig

2 Verwendung von Bibliotheken in UNIX-Systemen

- Großer Vorteil von dynamischen Bibliotheken:
  - Bibliotheken sind an einem zentralen Ort installiert (üblicherweise **`/usr/lib`**)
  - Bei einem Update (u. U. sicherheitskritisch!) muss nur eine Datei ausgetauscht werden
    - kein erneutes Binden aller betroffener Anwendungen nötig
- Komplett statisch gebundene Programme sind kaum mehr gebräuchlich
  - libc und andere Bibliotheken werden fast immer dynamisch gebunden
  - Manche Systeme (z. B. Solaris 10) enthalten gar keine statische libc mehr
- Ausnahmen:
  - Projektinterne Modularisierung (vgl. *sister*, *josh*)
  - Rescue-Binarys, Backup-Restore-Programme etc.
  - Eingebettete Systeme

U5-10 Aufgabe 4: jbuffer

1 Semaphor-Modul

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

2 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
- Nicht-blockierende Synchronisation der Konsumenten untereinander mittels CAS
  (siehe Vorlesung ab C|X-4, Seite 20)