Embedded Software Engineering 1

HS 2024 – Prof. Reto Bonderer

Autoren: Laurin Heitzer, Simone Stitz

https://github.com/P4ntomime/EmbSW1

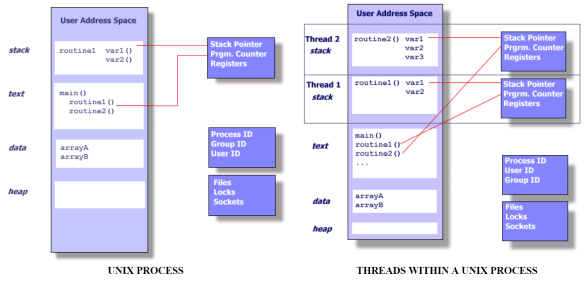
V1.0.20241115

Inhaltsverzeichnis

1 POSIX Threads Programming

Für UNIX Systeme steht ein stardardisiertes threads programming interface in C zur Verfügung (POSIX threads / pthreads).

1.1 UNIX Process vs. UNIX Thread



1.1.1 UNIX Process

- **heavyweight process** (generiert von Betriebssystem)
- Prozess erfordert **erheblichen overhead**, da Informationen über Programmressourcen und den Ausführungsstatus des Programms, beispielsweise:
  - Prozess-ID, Prozessgruppen-ID, Benutzer-ID und Gruppen-ID
  - Environment, Programmanweisungen
  - Register, Stack, Heap
  - Datei-Deskriptoren, Signal-Aktionen
  - Gemeinsame Bibliotheken
  - Werkzeuge für die prozessübergreifende Kommunikation

1.1.2 UNIX Thread

- lightweight 'process' (weniger overhead)
- Unabhängiger 'stream of instructions', welcher simultan mit anderen 'streams of instructions' ablaufen kann
- Prozedur, welche unabhängig von ihrem (aufrufenden) main-Programm abläuft
- **Threadsexistieren in einem Prozess und nutzen dessen Ressourcen**
  - Sobald ein Prozess ended, enden auch die darin existierenden Threads!
- **Ein Thread benutzt den gleichen Adressraum wie andere Threads im gleichen Prozess**
  - Daten können einfach mit anderen Threads im gleichen Prozess geteilt werden
- Threads werden vom Betriebssystem 'gescheduled'
- Ein Thread dupliziert nur die essenziellen Ressourcen die er braucht, um unabhängig 'schedulable' zu sein:
  - Stack pointer, Register
  - Scheduling properties (policy / priority)
  - Set of pendding and blocked signals
  - Thread-spezifische Daten

➔ **Gleichzeitigkeit wird in der Programmierung mit Threads umgesetzt!**

1.2 pthreads API

1.2.1 Includes / Compile & Link

- `#include <pthread.h>` wird benötigt
- Methoden der pthreads API starten mit `pthread_`
- Source files, welche pthreads verwenden, sollen mit `-pthread` kompiliert werden
- Für das file-linking muss der command `-lpthread` verwenet werden

Beispiel: Compiling / Linking file printer.c

Compiling: `clang -c -Wall -pthread printer.c`
Linking: `clang -o printer printer.o -Wall -lpthread`

1.2.2 Thread starten / beenden

- Jede Funktion mit der folgenden interface kann eine Thread-Methode werden
  - Als Parameter / Return-Wert sind alle Pointer-Datentypen möglich

  `void* threadRoutine(void* arg);`
- Ein Thread wird mit der folgenden Funktion gestartet:

```
int pthread_create(pthread_t* thread,              // ptr to pthread_t instance
                   const pthread_attr_t* attr,     // ptr to pthread_attr_t
                                                   // structure, often 0
                                                   // (default attributes)
                   void* (*startRoutine) (void*),  // function ptr to thread routine
                   void* arg);                     // single argument that may be
                                                   // passed to startRoutine
// returns 0 if thread is started successfully
```

- Ein Thread kann mit einer der folgenden drei Arten beendet werden
  - Thread ruft Funktion `pthread_exit()` auf
  - Thread springt aus Thread Routine `startRoutine` zurück
  - Thread wird mit Funktion `pthread_cancel()` abgebrochen

1.2.3 Warten, bis ein Thread beendet ist

- Nach dem Starten des Threads bzw. am Ende des main-Programms kann eine Endlos-Schleife eingefügt werden
  - **Dies sollte nie gemacht werden**, da der Prozess so die gesamten CPU-Ressourcen braucht
- Entsprechende Funktion aus pthreads API verwenden

```
int pthread_join(pthread_t thread,  // pthread_t instance
                 void** status)     // ptr to status argum. passed at end of thread
// returns 0 if thread terminated successfully
```

1.3 Beispiel: thread API

```
#include <pthread.h>  // for threads API
#include <stdio.h>
#include <unistd.h>   // for usleep()

// function prototype
void* printDashes(void* arg);

int main(void)
{
  int ret;
  pthread_t dasher; // pthread_t instance

  printf("start");

  // starts thread -> immediately returns
  // (thread maybe not fully started yet)
  ret = pthread_create(&dasher, 0,
                       printDashes, 0);
  if (ret)
  {
    printf("ERROR CODE: %d\n", ret);
    return -1;
  }

  // main thread shall wait until
  // dasher is finished
  ret = pthread_join(dasher, 0);
  if (ret)
  {
    printf("ERROR CODE: %d\n", ret);
    return -1;
  }

  printf("end\n");
  return 0;
}

void* printDashes(void* arg)
{
  for (size_t i = 0; i<20; ++i)
  {
    usleep(40000);
    putchar('-');
    fflush(stdout); // write character-
                    // wise and
                    // don't buffer
  }
  return 0;
}
```

1.4 Thread-safeness

Thread-safeness bezieht sich auf die Fähigkeit einer Anwendung, mehrere Threads gleichzeitig auszuführen, **ohne 'clubbering' und 'race conditions'** zu verursachen. Damit Thread-safeness gewährleistet werden kann, ist **Synchronisation** erforderlich.

clubbering: Speicher durcheinander bringen, wenn mehrere Threads den gleichen Speicher benötigen und 'falsch' darauf zugreifen

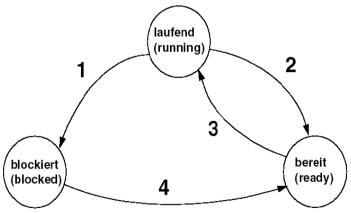
race conditions: Programmablauf und Endergebnis hängen davon ab, in welcher Reihenfolge 'gleichzeitig' ablaufende Threads auf z.B. eine globale Variable im Speicher zufreifen und das Verhalten somit unvorhersehbar wird

1.4.1 Empfehlung: Thread-Safeness

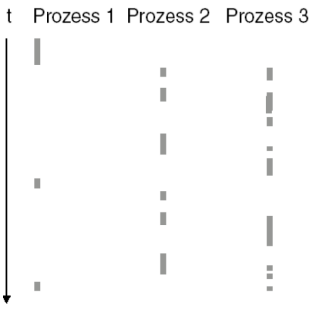
Wenn Thread-safeness nicht explizit garantiert ist (z.B. von einer Library, welche verwendet wird), muss angenommen werden, dass sie **nicht thread-safe** ist!
Um in einem solchen Fall Thread-safeness zu gewährleisten, können die Aufrufe einer 'unsicheren' Funktion **serialisiert** werden.

1.5 Quasi-Parallelität / 'Prozess'-Zustände

1.5.1 Prozess-Zustände



1. I/O Operation, Warten auf Bedingung
2. Scheduler entzieht CPU
3. Scheduler weist CPU zu
4. I/O beendet, Bedigung erfüllt



- Prozesse / Threads warten die 'meiste Zeit' ➔ blocked (z.B. `join` blockiert andere Threads)
- Scheduler ordnet CPU denjenigen Prozess / Thread zu, die im Zustand 'ready' sind und 'etwas zu tun haben'
- Die Zuordnung hängt vom verwendeten Scheduling-Algorithmus ab:
  - First come First serve Scheduling: Eine Queue mit allen Prozessen, wobei nächster Prozess jeweils hinten angehängt wird und erster Eintrag der Queue aktuell ausgeführt wird
  - Priority Scheduling: Pro Priorität gibt es eine Queue. Abarbeitung je nach Algorithmus anders

1.6 Synchronisation

Synchronisation wird benötigt, um den **Zugriff auf gemeinsame Ressourcen** in Critical Sections (CS) zu 'kontrollieren'.

1.6.1 Defintition: Critical Section (CS)

- Codebereich, in dem nebenläufige oder parallele Prozesse auf gemeinsame Ressourcen zugreifen
  - Zu jeder Zeit darf sich **höchstenns ein Prozess** im kritischen Abschnitt befinden
- Der Exklusive Zugriff durch höchstens einen Prozess wird mittels **gegenseitigem Ausschluss (Mutex)** sichergestellt ➔ Siehe Abschnitt 1.7

1.6.2 Forderungen an die Synchronisation

1. Maximal ein Prozess in einem kritischen Abschnitt (CS)
2. Über Abarbeitungsgeschwindigkeit, bzw. Anzahl Prozesse dürfen keine Annahmen getroffen werden
3. Kein Prozess darf **ausserhalb** eines kritischen Abschnitts einen anderen blockieren
4. Jeder Prozess, der am Eingang eines kritischen Absheritts wartet, muss irgendwann den Abschnitt betreten dürfen (**fairness condition**) ➔ Verhinderung von 'starvation'

1.7 Mutex (mutual exclusion)

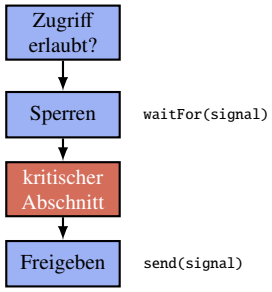
Die Lösungsstruktur 'Mutex' (gegenseitiger Ausschluss) stellt sicher, dass höchstens ein Prozess auf eine Critical Section (CS) zugreift.

1.7.1 Mutex – Ablauf

Zugriffsprüfung: Warten bis der Zugang frei wird

Sperren: Signal wird für andere auf Rot gesetzt, damit nur ein Prozess im kritischen Abschnitt sein kann

Freigeben: Rotes Signal wird wieder gelöscht



1.7.2 Verwendung von Signalen und Semaphoren

- Jeder Prozess wartet vor dem Betreten der CS auf ein gemeinsames Signal
  - Wenn das Signal gesetzt ist, ist CS frei
  - Mehrere Prozesse können gleichzeitig warten ➔ Schedulingalgorithmus bestimmt 'nächsten' Thread
- `waitFor(signal)` blockiert **aufrufenden** Prozess, falls Signal nicht gesetzt
- Jeder Prozess, der fertig ist, setzt das Signal mit `send(signal)`

Semaphoren:

- 'Semaphor' ist ein spezieller Name für ein Signal für den **Zutritt zu einer CS**
- Es gibt zwei atomare (nicht unterbrachbare) Operationen auf einer Semaphoren `s`
  - Passieren P(s): Beim Eintritt in CS ➔ `waitFor(s)`
  - Verlassen V(s): Beim Austritt aus CS ➔ `send(s)`

Bei der Verwendung von Semaphoren treten folgende Probleme auf

- Ressourcen können besetzt bleiben, wenn V(s) vergessen wird
  - **Für jedes P(s) braucht es auch ein V(s)**
- Grössere Programme: Es können subtile Probleme entstehen, falls z.B. das V(s) in einer **if-Bedingung** gemacht wird
- Beim Auftreten von Exceptions kann das Freigeben schwierig werden

➔ Lösung für das Freigabe-Problem: RAII (siehe Abschnitt)

1.7.3 Busy Waiting

- Prozesse warten **aktiv** in einer Schleife (**spin lock**)
  - Wartende Prozesse **belasten** unnötigerweise den Prozessor

Die Lösung für Busy Waiting ist, die wartenden Prozesse in eine **Warteschlange** einzutragen (**sleep and wakeup**)

1.8 Thread Synchronisierung in C mit pthreads API

Code Synchronisation wird mittels Mutex (**lock pattern**) sichergestellt. Das Konzept von Mutex ist, dass eine Mutex Variable **nur einem Thread gleichzeitig gehören kann.**

1.8.1 Ablaub einer Mutex-Sequenz in C

1. Mutex Variable erstellen / instanzieren
   - 'Schloss', welches Zugang zu CS schützt
2. Mehrere Threads versuchen, die Mutex Variable zu blockieren
   ➔ **Nur ein Thread** ist erfolgreich ➔ diesem Thread ('owner') gehört die Mutex Variable
3. Dieser 'owner thread' führt Aktionen in der Critial Section (CS) aus
   - Häufig Update einer globalen (shared) Variable
4. 'owner' entblockt (unlock) die Mutex Variable
5. Dem nächsten Thread gehört die Mutex Variable ➔ zurück zu Schritt 2
6. Wenn alle Threads abgearbeitet sind, wird die Mutex Variable zerstört

➔ Dies ist ein sicherer Weg, um sicherzustellen, dass, wenn **mehrere Threads** dieselbe Variable aktualisieren, der **Endwert derselbe** ist, wie wenn nur **ein Thread** die **Aktualisierung durchführen** würde.

Beispiel: Mutex in C

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* for usleep */

static volatile int val = 0;    // shared resource
static pthread_mutex_t valMtx;  // create mutex_t variable

void* threadRoutine(void* arg); // prototype

int main(void)
{
  pthread_t t1; // create pthread_t variable
  pthread_t t2; // create pthread_t variable

  pthread_mutex_init(&valMtx, 0); // init mutex

  pthread_create(&t1, 0, threadRoutine, 0);
  pthread_create(&t2, 0, threadRoutine, 0);

  pthread_join(t1, 0);  // wait for thread to finish
  pthread_join(t2, 0);  // wait for thread to finish

  pthread_mutex_destroy(&valMtx); // destory mutex

  return 0;
}

```

```c
// main routine of each counter thread
void* threadRoutine(void* arg)
{
  unsigned int rState = 17;

  while(1)
  {
    /* non critical section; simulate with usleep() */
    usleep(rand_r(&rState) % 200000);

    /* start of critical section */
    pthread_mutex_lock(&valMtx);  // lock mutex

    if (val < 20)
    {
      /* wait random time between 0s up to 0.3s */
      usleep(rand_r(&rState) % 300000);
      val = val + 1;  // change shared resource
      printf("val = %2d\n", val);
    }
    else
    {
      /* end of critical section */
      pthread_mutex_unlock(&valMtx);  // unlock mutex
      break;                          // exit while(1)
    }
    /* end of critical section */
    pthread_mutex_unlock(&valMtx);    // unlock mutex
  }
  pthread_exit(0); // optional, good programming style!
}
```

1.9 Monitorprinzip (Monitor Pattern)

Das Monitorprinzpt beschreibt eine Art Abstraktion des Mutex / Lock Patterns.
Dabei muss sich der **Aufrufer nicht mehr um die Synchronisation der Threads kümmern**. Das Problem wird einmal im Monitor gelöst.

- Es wird ein Abstrakter Datentyp (ADT) definiert, der genau die Funktionen in der Schnittstelle anbietet, die notwendig sind
- Der Aufrufer ruft diese Funktion auf, muss sich aber **nicht um Synchronisation kümmern**
  - Synchronisation (z.B. mit Semaphoren) ist Implementation des Monitors lokal gelöst

1.10 'Stolperfallen' bei Synchronisation

1.10.1 Starvation (Verhungern)

- Zustand, bei dem ein Prozess nie dran kommt ➔ er verhungert
- Kann auftreten bei:
  - prioritätsgetriebenen Systemen bei Prozessen mit niederer Priorität passieren
  - SJF (shortest job first) Systeme ➔ kurze Jobs bremsen längere Jobs aus
- Fairness condition besagt, dass Starvation verhindert werden muss

1.10.2 Deadlock

- Situation, bei der sich **zwei Prozesse gegenseitig blockieren**
  - Zwei Prozesse benötigen gemeinsame Ressourcen A und B. Wenn Prozess 1 die Ressource A bereits besitzt und Prozess 2 die Ressource B, dann warten beide unendlich lange auf die jeweils andere Ressource

Deadlock kann vermieden werden, indem alle Prozesse die gemeinsamen Ressourcen immer in derselben Reihenfolge anfordern (z.B. zuerst A, dann B)