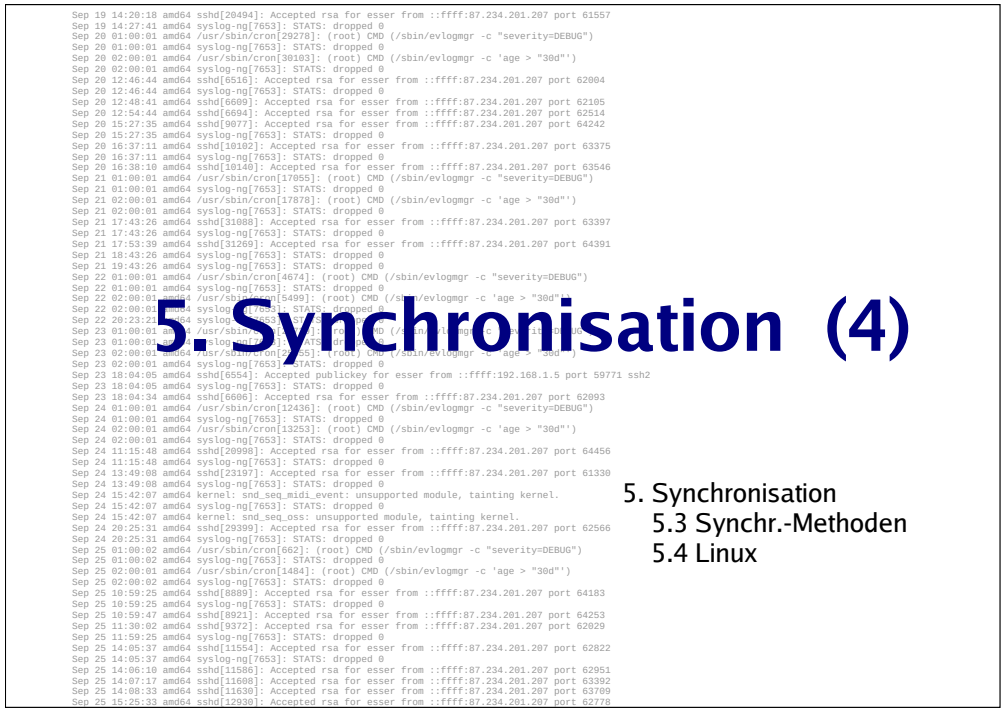


5. Synchronisation (4)

5. Synchronisation
   5.3 Synchr.-Methoden
   5.4 Linux

Nachrichten (1)

- Nachrichtenaustausch über zwei Systemaufrufe
  - `send (destination, &message);`
  - `receive (source, &message);`
- Synchrone Kommunikation:
  Threads blockieren, wenn *send* bzw. *receive* nicht sofort ausgeführt werden können, z. B. weil
  - die Gegenseite keinen entsprechenden Befehl abgesetzt hat,
  - ein Zwischenpuffer für die Nachrichten voll bzw. leer ist.

Nachrichten (2)

- Vorteil: funktioniert auch bei Systemen ohne gemeinsamen Hauptspeicher (distributed systems, client-server-computing)
- Nachteile:
  - aufwändiger durch Duplizieren der Daten
  - Verwaltung der Namen für Quelle und Ziel nötig
  - Vorkehrungen gegen Verlust der Meldung nötig
- Implementierung z. B. durch Pipes oder Mailslots (Windows) oder RPCs.

Nachrichten (3)

- **synchron** vs. **asynchron**
  - synchron: *send* / *receive* blockieren, bis zugehörige Operation auf Gegenseite abgeschlossen ist
  - asynchron: *send*-Call kehrt sofort zurück; Erfolg des Versands ist evtl. überprüfbar, z. B.:
    - Gegenseite schickt explizit Antwort
    - Messaging-System sendet Signal bei Zustellung
- **verbindungsorientiert** vs. **verbindungslos**
  - verbindungsorientiert: „stehende Verbindung“ (TCP)
  - verbindungslos (vgl. UDP)

Beispiele für Nachrichten (1)

Zwei Prozesse wechseln sich im Zugriff ab

```
void funktion (int id) {
  int otherid = 1 - id;
  char message[10] = "";
  // ein Prozess darf zuerst; ID 0
  if (id==0) {
    message = "go";
  }
  // unkritische Befehle
  while message != "go" {
    receive (otherid, &message);
  }
  // kritischer Bereich
  send (otherid, "go");
  message = "";
  // mehr unkritische Befehle
}
```

p0 ruft *funktion(0)* auf,
p1 ruft *funktion(1)* auf.

p0 darf zuerst in
den kritischen Bereich

Beispiele für Nachrichten (2)

Server-Prozess gestattet Zugriff auf Ressource

```
//--- server ------------------

int busy;  // globaler Zustand

void main () {
  initialize ();
  while TRUE {
    receive (&id, &message);
    // Neben Nachricht auch
    // Absender (id) bekannt
    switch (message) {
      "request": enter_in_queue(id);
      "release": let_next_one();
    };
  };
};
```

```
void let_next_one () {
  if queue_is_empty () { busy = FALSE; }
  else {
    id = queue_head.id;
    send (&id, "go");
    queue_head = queue_head.next;
  };
}

void enter_in_queue (int id) {
  if queue_is_empty () and busy==FALSE {
    busy = TRUE;
    send (&id, "go")
  } else {
    allocate (&newentry);
    newentry.id = id;
    newentry.next = NULL;
    queue_last.next = newentry;
    queue_last = newentry;
  }
}
```

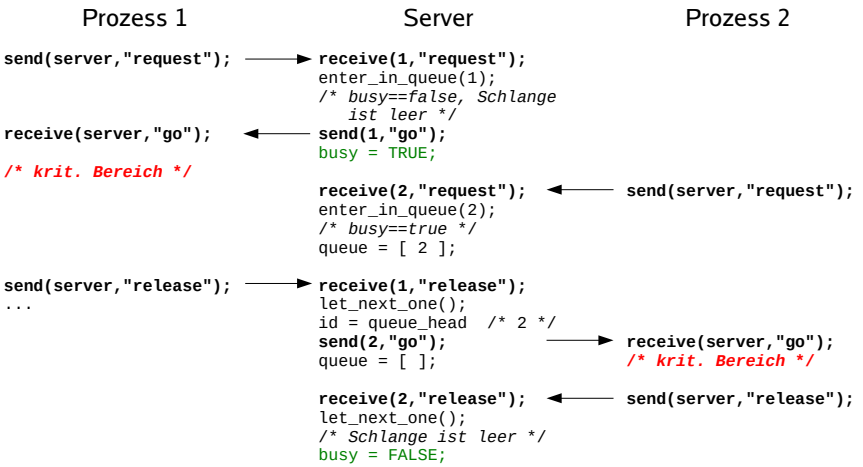

Beispiele für Nachrichten (3)

Client fordert Zugriffsberechtigung beim Server an

```
//--- client ------------------

void funktion (int id) {
  reply = "";
  // unkritische Befehle
  while (reply != "go") {
    send (server, "request");
    receive (server, &reply);  // auf Freigabe warten
  };
  // kritischer Bereich
  send (server, "release");    // krit. Region freigeben
  // mehr unkritische Befehle
}
```


Beispiele für Nachrichten (4)

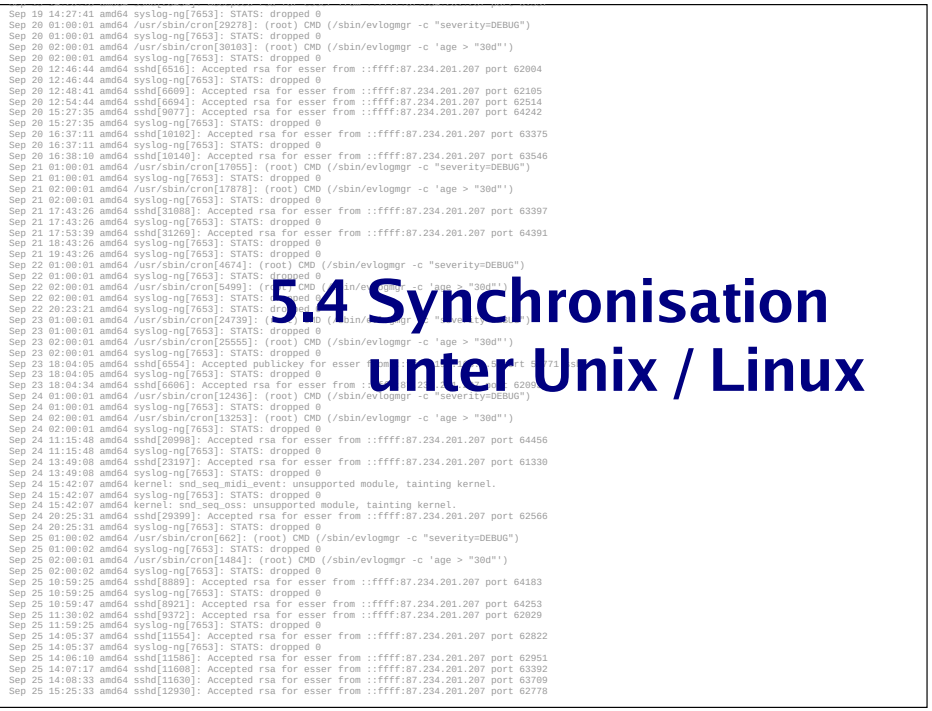


Nachrichten

- Auch Broadcast an mehrere Prozesse möglich
- Verteilte Systeme: Voting-Verfahren (über Broadcast) zum Fällen von Entscheidungen

Mehr zu Nachrichten:

Kapitel 6: **IPC (Inter Process Communication)**



5.4 Synchronisation unter Unix / Linux

Inhaltsübersicht

POSIX Threads

POSIX-Threads können verschiedene Standard-Synchronisations-Primitive verwenden:

- Mutexe
- Semaphore
- Bedingungsvariablen

Mutexe in POSIX (1)

pthread_mutex_init

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

- erzeugt einen neuen Mutex mit bestimmten Attributen (attr kann auch NULL sein)
- Initialisierung im Hauptprogramm, bevor die verschiedenen Threads starten und den Mutex nutzen
- „Abkürzung“:
  `pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;`

Mutexe in POSIX (2)

pthread_mutex_lock

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

- pthread_mutex_lock
  - versucht, das Lock zu erhalten
  - blockiert, wenn das Lock schon vergeben ist
- pthread_mutex_trylock
  - versucht ebenfalls, das Lock zu erhalten
  - Aufruf kehrt auch bei Misserfolg zurück (mit Fehlercode **EBUSY**)

Mutexe in POSIX (3)

pthread_mutex_unlock

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

- Sperre wieder aufheben

pthread_mutex_destroy

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Mutex nicht mehr verwenden

Mutexe in POSIX (4)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 =
  PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
#define THREAD_ZAHL 2

main() {
   int rc[THREAD_ZAHL];
   int i;
   pthread_t thread[THREAD_ZAHL];

   /* Zwei Threads erzeugen, die functionC() ausführen */
   for (i=0; i<THREAD_ZAHL; i++) {
     if( (rc[i]=pthread_create( &thread[i], NULL, &functionC, NULL)) ) {
       printf("Thread-Erzeugung fehlgeschlagen: %d\n", rc[i]);
     };
   };
   /* Auf die beiden Threads warten */
   for (i=0; i<THREAD_ZAHL; i++) pthread_join( thread[i], NULL);
   printf("Endergebnis: %d\n",counter);
   exit(0);
}
```

```
void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Mutexe in POSIX (5)

- ... und in broken_threads.c einmal mit auskommentierten Mutexen (kritischer Bereich ungeschützt):

```
void *functionC()
{
   // pthread_mutex_lock( &mutex1 );
   int tmp=counter;                 // gemeinsame Variable auslesen
   for (i=0; i<999999; i++) {};     // etwas Zeit verbrauchen ...
   tmp++;
   counter=tmp;                     // gemeinsame Variable zurückschreiben
   printf("Counter value: %d\n",counter);
   // pthread_mutex_unlock( &mutex1 );
}
```


Mutexe in POSIX (6)

Test mit 20 Threads

```
$ gcc -lpthread -o threads threads.c
$ gcc -lpthread -o broken_threads broken_threads.c
```

```
$ ./pthread
Counter value: 1
Counter value: 2
Counter value: 3
Counter value: 4
Counter value: 5
Counter value: 6
Counter value: 7
Counter value: 8
Counter value: 9
Counter value: 10
Counter value: 11
Counter value: 12
Counter value: 13
Counter value: 14
Counter value: 15
Counter value: 16
Counter value: 17
Counter value: 18
Counter value: 19
Counter value: 20
Endergebnis: 20
```

```
$ ./broken_threads
Counter value: 1
Counter value: 2
Counter value: 3
Counter value: 3
Counter value: 3
Counter value: 4
Counter value: 5
Counter value: 6
Counter value: 7
Counter value: 3
Counter value: 4
Counter value: 5
Counter value: 6
Counter value: 8
Counter value: 7
Counter value: 7
Counter value: 8
Counter value: 8
Counter value: 7
Counter value: 7
Endergebnis: 7
```


Mutexe in POSIX (7)

Mutexe wahlweise „rekursiv“:

normal	rekursiv

`pthread_mutex_lock (mutex);`
`/* Code */`
`pthread_mutex_lock (mutex);`
`/* Aufruf blockiert, mutex schon gesperrt */`
`pthread_mutex_unlock (mutex);`
`pthread_mutex_unlock (mutex);`
	`pthread_mutex_lock (mutex);`
	`/* Code */`
	`pthread_mutex_lock (mutex);`
	`/* erfolgreich, weil gleicher Thread dieselbe Sperre anfordert */`
	`pthread_mutex_unlock (mutex);`
	`pthread_mutex_unlock (mutex);`

rekursiv:

```
pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```


Semaphore in POSIX (1)

sem_init

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- erzeugt einen neuen Semaphor mit Startwert *value*
- Initialisierung im Hauptprogramm, bevor die verschiedenen Threads starten und den Semaphor nutzen
- *pshared*: für gemeinsame Nutzung durch mehrere Prozesse (geht unter Linux nicht)

Semaphore in POSIX (2)

sem_wait, sem_trywait

```
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
```

- sem_wait implementiert die wait()-Operation
  - erniedrigt den Zähler c im Semaphor, falls c>0
  - blockiert den Thread, bis c>0 ist
- sem_trywait
  - erniedrigt den Zähler c im Semaphor, falls c>0
  - gibt Fehlerwert EAGAIN zurück, falls c<=0

Semaphore in POSIX (3)

sem_post

```
int sem_post(sem_t * sem);
```

- sem_post implementiert die signal()-Operation
  - erhöht den Zähler c im Semaphor
  - blockiert *nie*

sem_destroy

```
int sem_destroy(sem_t * sem);
```

- Semaphor nicht mehr verwenden

Semaphore in POSIX (4)

sem_getvalue

```
int sem_getvalue(sem_t * sem, int * sval);
```

- sem_getvalue liest den Wert des Semaphors aus und schreibt ihn in die angeg. Variable

Semaphore in POSIX (5)

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

static sem_t sem;
void *functionC();
int counter = 0;
#define THREAD_ZAHL 2

main() {
   int rc[THREAD_ZAHL]; int i;
   pthread_t thread[THREAD_ZAHL];
   /* Semaphor auf 1 initialisieren */
   sem_init(&sem, 0, 1);
   /* Zwei Threads erzeugen, die functionC() ausführen */
   for (i=0; i<THREAD_ZAHL; i++) {
     if( (rc[i]=pthread_create( &thread[i], NULL, &functionC, NULL)) ) {
       printf("Thread-Erzeugung fehlgeschlagen: %d\n", rc[i]);
     };
   };
   /* Auf die beiden Threads warten */
   for (i=0; i<THREAD_ZAHL; i++) pthread_join( thread[i], NULL);
   printf("Endergebnis: %d\n",counter);
   exit(0);
}

void *functionC() {
   int i;
   sem_wait( &sem );
   int tmp=counter; // Variable auslesen
   for (i=0; i<999999; i++) {};
      // etwas Zeit verbrauchen ...
   tmp++;
   counter=tmp;
      // gemeinsame Variable zurückschreiben
   printf("Counter value: %d\n",counter);
   sem_post( &sem );
}
```

POSIX-Bedingungsvariablen (1)

- Idee: Threads warten darauf, dass eine bestimmte Bedingung erfüllt ist, und schlafen solange (vgl. *Monitore*)
- Zwei Basisfunktionen:
  - **pthread_cond_signal** & **thread_cond_broadcast**
    (Erfülltsein der) Bedingung signalisieren
    ➞ weckt alle Threads auf, die darauf warten
    (nicht schlafende Threads erhalten kein Signal)
  - **pthread_cond_wait**
    warten, bis die Bedingung erfüllt ist
- Bedingungsvariablen immer zusammen mit Mutex

POSIX-Bedingungsvariablen (2)

pthread_cond_init

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

- erzeugt eine neue Bedingungsvariable
- Initialisierung im Hauptprogramm, bevor die verschiedenen Threads starten und die Bedingungsvariable nutzen
- „Abkürzung“:
  `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`

POSIX-Bedingungsvariablen (3)

pthread_cond_wait

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

- pthread_cond_wait entsperrt den Mutex und wartet darauf, dass die Bedindung cond signalisiert wird (Thread schläft)
- Wird die Bedingung signalisiert, sperrt die Funktion den Mutex, bevor sie die Kontrolle an den aufrufenden Thread zurückgibt

POSIX-Bedingungsvariablen (4)

pthread_cond_signal & pthread_cond_broadcast

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- pthread_cond_*signal* weckt ***einen der*** Threads, die auf die Bedingung *cond* warten. (Wartet niemand, passiert nichts.)
- pthread_cond_*broadcast* weckt ***alle*** Threads, die auf die Bedingung *cond* warten. (Wartet niemand, passiert nichts.)

POSIX-Bedingungsvariablen (5)

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
  // Mutex schützt Zugriff auf x, y
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
  // Bedingung: x > y

thread_eins () {
  /* warten auf x > y */
  pthread_mutex_lock(&mut);
  while (x <= y) {
    pthread_cond_wait(&cond, &mut);
  }
  // mit x und y arbeiten
  pthread_mutex_unlock(&mut);
}

thread_zwei () {
  pthread_mutex_lock(&mut);
  // x und y verändern
  if (x > y) pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mut);
}
```


Übersicht der POSIX-Funktionen

	Mutexe	Semaphore	Bedingungsvariablen
Warten bzw. Blockieren	pthread_mutex_lock, pthread_mutex_trylock	sem_wait, sem_trywait	pthread_cond_wait
Signalisieren	pthread_mutex_unlock	sem_post	pthread_cond_signal, pthread_cond_broadcast
Erzeugen	pthread_mutex_init	sem_init	pthread_cond_init
Zerstören	pthread_mutex_destroy	sem_destroy	pthread_cond_destroy

Vorschau

Nächstes Mal:

Synchronisation von Prozessen;
Synchronisation im Linux-**Kernel**