Programmiertechnik (C++)

Prof. Dr. Stefan Enderle
NTA Isny

8. Interprozesskommunikation (IPC)

7.1 Einführung

- Unterscheidung:
  - Programm
  - Prozess
  - Thread

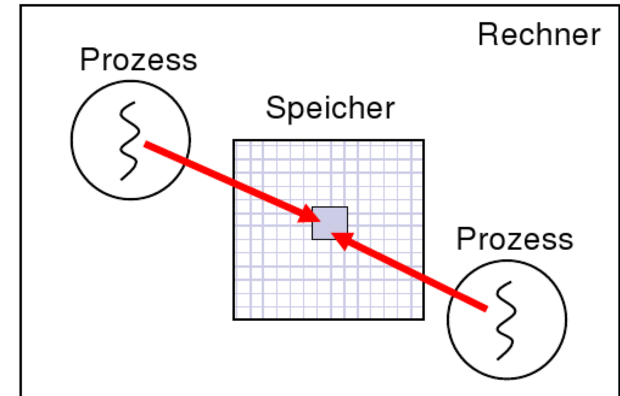
- Im Weiteren „Prozess“
- Wie kann ein Prozess Daten mit einem anderen Prozess austauschen?
  - Über gemeinsame Variablen
  - Über Nachrichtenaustausch

Fragen

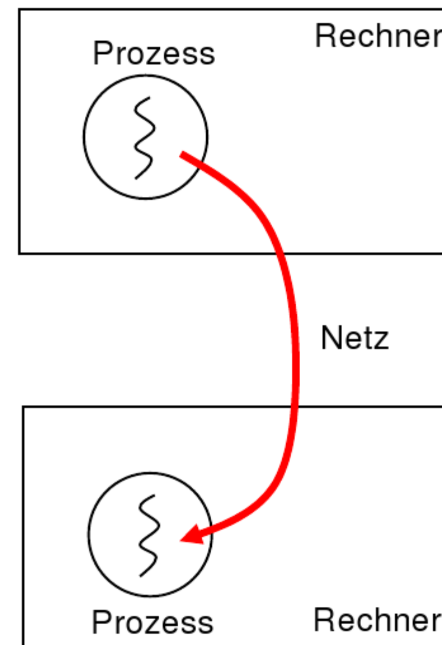
- Wie kann ein Prozess Daten mit einem anderen Prozess austauschen?
- Wie wird sichergestellt, dass Prozesse nicht gleichzeitig auf gemeinsame Information zugreifen?
- Wie wird die richtige Reihenfolge des Zugriffs sichergestellt?

Kommunikationsformen

- *„Shared Memory“*
  - Gemeinsame Variablen
  - In Ein-Prozessor-Systemen oder Mehr-Proz.-S. mit gemeinsamem physik. Speicher



- Nachrichtenaustausch
  - v.a. bei veteilten Systemen, also über Rechnergrenzen hinweg



Erzeuger-Verbraucher-Problem

- Problemstellung:
  - Zwei Prozesse besitzen einen gemeinsamen Puffer fester Länge.
  - Ein Prozess schreibt Informationen in den Puffer.
  - Der andere liest aus dem Puffer.

Erzeuger-Verbraucher-Problem

- Erzeuger:

```
while (true) {
  produce(item);          // Element erzeugen
  insert(item);           // in Puffer schreiben
}
```

- Verbraucher:

```
while (true) {
  remove(item);           // Element aus Puffer holen
  consume(item);          // und verbrauchen
}
```

Erzeuger-Verbraucher-Problem

- Problem:
  - Der Erzeuger darf nicht in den vollen Puffer einfügen.
  - Der Verbraucher darf nicht aus dem leeren Puffer lesen.

  → Beide müssen also unter Umständen warten.

Erzeuger-Verbraucher-Problem

- **Implementierung:**
  - „Ringpuffer“
  - Puffergröße: `N`
  - Füllstand: `count`
  - Insert Position: `in`
  - Remove Position: `out`

Erzeuger-Verbraucher-Problem

- ```
  void insert(Item item)
  {
    while (count==N) sleep(1);
    buffer[N] = item;
    in = (in+1)%N;
    count = count+1;
  }
  ```
- ```
  Item remove()
  {
    while (count==0) sleep(1);
    item = buffer[out];
    out = (out+1)%N;
    count = count-1;
    return item;
  }
  ```

Erzeuger-Verbraucher-Problem

- **Synchronisationsproblem:**
  - Die Anweisungen
    `count = count+1` bzw. `count = count-1`
    werden z.B. in folgende Maschinenbefehle übersetzt:

```
P1: register1 = count
P2: register1 ++
P3: count = register1

C1: register2 = count
C2: register2 --
C3: count = register2
```

  - Sei `count = 5`.
    Was passiert bei Ausführung in folgender Reihenfolge:
    - P1, P2, C1, C2, P3, C3
    - P1, P2, C1, C2, C3, P3

Race-Condition

- **„Race-Condition“:**
  - Werden gemeinsame Daten von mehreren Prozessen verändert, kann es zu einer Race-Condition kommen.
  - Hierbei hängt der Ausgang von der Bearbeitungsreihenfolge bzw. vom Zeitpunkt des Task-Switches ab.

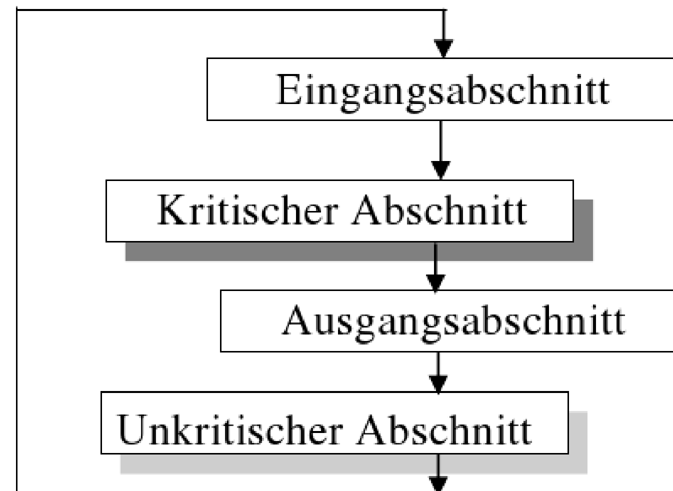
- Abhilfe: Synchronisationsverfahren

7.2 Synchronisationsverfahren

- **Kritischer Abschnitt (critical section):**

  Menge von Instruktionen, in der das Ergebnis der Ausführung auf unvorhergesehene Weise variieren kann, wenn Variablen, die auch für andere parallel ablaufende Prozesse zugreifbar sind, während der Ausführung verändert werden.

- Prinzipieller „Lebenszyklus“ eines Prozesses oder Threads:



Synchronisationsverfahren

- Gesucht:
  - „Protokoll“, an das sich alle Prozesse halten und das die Semantik des kritischen Abschnitts realisiert.

- Anforderungen an eine Lösung:
  - Zwei Prozesse dürfen nicht gleichzeitig in ihrem kritischen Abschnitt sein (safety).
  - Es dürfen keine Annahmen über die Bearbeitungs- geschwindigkeit von Prozessen gemacht werden.
  - Kein Prozess, der außerhalb eines kritischen Bereichs ist, darf andere Prozesse beim Eintritt in den kritischen Abschnitt behindern.
  - Kein Prozess darf ewig auf den Eintritt in den kritischen Abschnitt warten müssen (fairness).
  - Möglichst passives statt aktives Warten.

Synchronisationsverfahren

- Lösungen für die Synchronisation:
  - Semaphore
  - Mutexe
  - Monitore

- Bemerkung:

  Diese Lösungen synchronisieren die Nutzung gemeinsamer Variablen. Bei Nachrichtenkommunikation wird diese Form der Synchronisation nicht benötigt.

Semaphore

- Ein Semaphor ist eine geschützte Variable, auf die nur mit zwei atomaren Operationen zugegriffen werden kann:
  - up (auch „signal“ oder „V“)
  - down (auch „wait“ oder „P“)

- Beispiel-Implementierung:
  - Aufruf vor Eintritt in einen kritischen Abschnitt:

```
down(s)
{  s = s-1;
   if (s<0) queue_this_process_and_block();
}
```

  - Aufruf nach Austritt aus kritischem Abschnitt:

```
up(s)
{  s=s+1;
   if (s>=0) wakeup_process_from_queue();
}
```

Semaphore

- **Eigenschaften:**
  - Semaphore können zählen und damit z.B. die Nutzung gemeinsamer Betriebsmittel überwachen.
  - Semaphore werden durch spezielle Systemaufrufe implementiert, die die geforderten atomaren Operationen up und down realisieren.
  - Semaphore können in beliebigen Programmiersprachen benutzt werden, da sie letztlich einem Systemaufruf entsprechen.
  - Semaphore realisieren ein passives Warten bis zum Eintritt in den kritischen Abschnitt.

Mutex

- **Eigenschaften:**
  - Oft wird die Fähigkeit zu zählen bei Semaphoren nicht benötigt, d.h., es genügt eine einfache binäre Aussage, ob ein kritischer Abschnitt frei ist oder nicht.
  - Dazu kann eine einfacher zu implementierende Variante, der sogenannte Mutex (von „mutual exclusion“), verwendet werden.

Erzeuger-Verbraucher-Problem mit Semaphoren

- ```
  void insert(Item item)
  {
    while (count==N) sleep(1);
    buffer[N] = item;
    in = (in+1)%N;
    count = count+1;
  }
  ```
- ```
  Item remove()
  {
    while (count==0) sleep(1);
    item = buffer[out];
    out = (out+1)%N;
    count = count-1;
    return item;
  }
  ```

Erzeuger-Verbraucher-Problem mit Semaphoren

- **Schutz:**
  1 Produzent – 1 Verbraucher

- Start mit folgenden Semaphoren:
  `mutex = 1;`

- ```
  produce(item);
  down(mutex);
  add(item);
  up(mutex);
  ```

- ```
  down(mutex);
  remove(item);
  up(mutex);
  consume(item);
  ```

Erzeuger-Verbraucher-Problem mit Semaphoren

- **Schutz:**
  Mehrere Produzenten - mehrere Verbrauchern

- Start mit folgenden Semaphoren:
  ```
  mutex = 1;
  empty = N;
  full  = 0;
  ```

- ```
  produce(item);
  down(empty);
  down(mutex);
  add(item);
  up(mutex);
  up(full);
  ```

- ```
  down(full);
  down(mutex);
  remove(item);
  up(mutex);
  up(empty);
  consume(item);
  ```

Implementierung von Semaphoren

- Immer als Systemaufruf des Betriebssystems.
- Ein-Prozessor-System:
  Es werden kurzzeitig sämtliche Unterbrechungen (Interrupts) verhindert.
- Bei Mehrprozessor-Systemen:
  Mittels atomarer Prozessor-Operation
- Beispiel: TSL Befehl (Test-and-set-lock) sperrt Memory Bus für andere Operationen

Semaphore in Qt

- Klasse: `QSemaphore` (`#include <QtCore>`)

```
QSemaphore sem(5);          // sem.available() == 5

sem.acquire(3);             // sem.available() == 2
sem.acquire(2);             // sem.available() == 0
sem.release(5);             // sem.available() == 5
sem.release(5);             // sem.available() == 10

sem.tryAcquire(1);          // sem.available() == 9,
                               returns true
sem.tryAcquire(250);        // sem.available() == 9,
                               returns false
```

Mutexe in Qt

- Klasse: `QMutex` (`#include <QMutex>`)

```
QMutex mutex;

mutex.lock();     // Eintritt in krit. Bereich
...
mutex.unlock();   // Austritt aus krit. Bereich
```

Mutexe in Qt

Beispiel: Ohne gegenseitigen Ausschluss:

```
int number = 6;

void method1()
{
  number *= 5;
  number /= 4;
}

void method2()
{
  number *= 3;
  number /= 2;
}
```

Mutexe in Qt

Beispiel: Mit gegenseitigem Ausschluss:

```
QMutex mutex;
int number = 6;

void method1()
{
    mutex.lock();
    number *= 5;
    number /= 4;
    mutex.unlock();
}
```

```
void method2()
{
    mutex.lock();
    number *= 3;
    number /= 2;
    mutex.unlock();
}
```

Mutexe in Qt

Weitere Vereinfachung: Mutex-Locker:

Klasse: `QMutexLocker`

```
QMutex mutex;
int number = 6;
```

```
void method1()
{
 QMutexLocker locker(&mutex);
 number *= 5;
 number /= 4;
}
```

```
void method2()
{
   QMutexLocker locker(&mutex);
   number *= 3;
   number /= 2;
}
```