Übungsbeispiel 3

Explizite Synchronisation & Kommunikation über Shared Memory

LU 182.023 Systemnahe Programmierung

Christian El Salloum

Überblick

- Organisatorisches
- Shared Memory (Kommunikation)
- Synchronisation von mehreren Prozessen
- Synchronisationskonstrukte
  - Semaphore
  - Sequencer und Eventcounts
- Ressourcenverwaltung
- Zusammenfassung

Interaktion von Prozessen in Bsp 3 (Kommunikation und Synchronization)

- Shared Memory (Kommunikation)
- Synchronisation von mehreren Prozessen
- Synchronisationskonstrukte
  - Semaphore
  - Sequencer und Eventcounts

Shared Memory (1/4)

- Vier Operationen:
  - Anlegen
    ```
    int shmget(key_t key, size_t size, int flags);
    ```
  - Einhängen
    ```
    void *shmat(int shmid, const void *shmaddr,
                int shmflg);
    ```
  - Aushängen
    ```
    int shmdt(const void *addr);
    ```
  - Löschen
    ```
    int shmctl(int shmid, int cmd,
               struct shmid_ds *buf);
    ```


Shared Memory (2/4)

```
typedef struct shm_struct {
    int state;
    int data[MAX_DATA];
} SHMSTRUCT;
…

int main(int agrc, char **argv) {
    …
    /* Anlegen von Shared Memory */
    if ((shmid = shmget (KEY, sizeof(SHMSTRUCT),
                IPC_CREAT | PERMISSION)) < 0) {
        /* Fehlerbehandlung */
        …
    }
```


Shared Memory (3/4)

```
/* Einhängen von Shared Memory */
if ((shmp = (SHMSTRUCT *)shmat(smid,NULL,0) ==
          (SHMSTRUCT *) -1) {
    /* Fehlermeldung */
    …
}
do {
    /* Synchronisierter Zugriff auf Shared memory */
    …
    shmp -> data[0] = …;    /* write */
    if (shmp -> state == ENDSIGNAL) { Ende = 1; }
    …
}
while (Ende == 0);
```

Shared Memory (4/4)

```
/* Aushängen von Shared Memory */
if (shmdt ((void *)shmp) < 0) {
     /* Fehlerbehandlung */
     …
}
/* Löschen von Shared Memory */
if (shmctl(shmid, IPC_RMID, NULL) < 0) {
     /* Fehlerbehandlung */
     …
}
```


Synchronisation von Prozessen (1/2)

- *Kritischer Abschnitt (k.A.)*
  - Ein Prozess befindet sich im k.A., wenn er auf gemeinsame Daten (*shared memory*) zugreift
- *Mutual Exclusion*
  - Zu jeden Zeitpunkt darf sich maximal ein Prozess in seinem kritischen Abschnitt befinden
  - Eintritt muss geregelt erfolgen

Synchronisation von Prozessen (2/2)

- *Beliebige Ausführungsreihenfolge*
  - Zugriffsreihenfolge auf Shared Memory "beliebig", darf Konsistenz nicht verletzen!
  - Beispiel: Reservierungssystem
- *Vorgegebene Ausführungsreihenfolge*
  - Zugriffe der Prozesse auf Shared Memory müssen in vorgegebener Reihenfolge erfolgen
  - Beispiel: Prozess A schreibt Daten auf Shared Memory, Prozess B liest diese Daten vom Shared Memory, dann wiederum schreibt Prozess A neue Daten...

Semaphore - Funktionalität

- "*Gemeinsame Variable*" S zur Synchronisation
- Operation `Wait(S)` bzw. `P(S)`:
  - Prozess darf in k.A. eintreten, wenn Semaphore > 0, sonst wird er blockiert
  - Kein *busy waiting*, sondern Blockieren des Prozesses durch das Betriebssystem
  - Prozess tritt in k.A. ein⇒dekrementiert Semaphore
- Operation `Signal(S)` bzw. `V(S)`
  - Prozess verlässt k.A.⇒inkrementiert Semaphore

Semaphore - Operationen

- `#include <sem182.h>  /* http://www.vmars.tuwien.ac.at/download/sem182.tgz */`
- Anlegen
  `int seminit(key_t key, int semperm, int initval);`
- oder Holen
  `int semgrab(key_t key);`
- Operationen
  `int P(int semid);`
  `int V(int semid);`
- Löschen
  `int semrm(int semid);`
- Compilieren
  `c89 -… -o prg prg.c -lsem182`

Beliebige Zugriffsfolge mit Semaphoren

```
/* Process A          */      /* Process B          */
s=seminit(KEY1,PERM,1);        s=semgrab(KEY1);

for(;;)                        for(;;)
{                              {
   if (P(S)< 0)                   …P(S)…
     /*Fehlerbehandlung*/
   critical();                    critical();
   …V(S)…                         …V(S)…
}                              }
```

Abwechselndes Abarbeiten

```
/* Process A          */    /* Process B          */
s1=seminit(KEY1,PERM,1);     s1=semgrab(KEY1);
s2=seminit(KEY2,PERM,0);     s2=semgrab(KEY2);

for(;;)                      for(;;)
{                            {
   …P(S1)…                      …P(S2)…
   critical();                  critical();
   …V(S2)…                      …V(S1)…
}                            }
```


Eventcount - Funktionalität

- Eventcount E ist ganzzahlige Variable zum Zählen von Vorkommnissen von Ereignissen
  Der Anfangswert ist 0.
- Operation `await(E,v)`
  - Prozess darf in k.A. eintreten, wenn E >= v, sonst wird er blockiert (kein *busy waiting*)
- Operation `advance(E)`
  - P verlässt k.A.⇒inkrementiert Wert des Eventcount E

Eventcount - Operationen

- `#include <sequev.h>`
- Anlegen
  `eventcounter_t *create_eventcounter(key_t key);`
- Warten
  `int eawait(eventcounter_t *evc, long value);`
- Erhöhen
  `int eadvance(eventcounter_t *evc);`
- Wert lesen
  `long eread(eventcounter_t *evc);`
- Löschen
  `int rm_eventcounter(eventcounter_t *evc);`

Abwechselndes Abarbeiten mit Eventcounts

```
/* Process A          */    /* Process B          */

long t;                      long t;
e=create_eventcounter(K1); e=create_eventcounter(K1);

t = 0;                       t = 1;
for(;;)                      for(;;)
{                            {
   …eawait(e,t)…                …eawait(e,t)…

   critical();                  critical();

   …eadvance(e)…                …eadvance(e)…
   t += 2;                      t += 2;
}                            }
```


Initialisierung von Eventcount

Aufgabe 1		Aufgabe 2	
Evc	Prozess	Evc	Prozess
0	A	0	B
1	B	1	A
2	B	2	A
3	A	3	A
4	B	4	B
5	B	5	A
6	A	6	A
7	B	7	A
8	B	8	B
9	A	9	A
10	B	10	A

gegeben:
Prozess A und B
gesucht:
t=? t=t+?

Aufgabe 1:

```
A: t=0;
t=t+3;
B: t=1
if ((t%3)==2)
  t=t+2;
else
  t=t+1;
```

Aufgabe 2:
...

Beliebige Zugriffsfolge mit Eventcounts ?!

```
/* Process A          */    /* Process B          */

long t;                      long t;
e=create_eventcounter(K1); e=create_eventcounter(K1);

t = ?;                       t = ?;
for(;;)                      for(;;)
{                            {
   …eawait(e,t)…                …eawait(e,t)…

   critical();                  critical();

   …eadvance(e)…                …eadvance(e)…
   t = ?;                       t = ?;
}                            }
```

Sequencer - Funktionalität

- Sequencer S ist ganzzahlige Variable mit Anfangswert 0, die verwendet wird, um die Abfolge von Ereignissen zu entscheiden.
- Operation `ticket(S)`
  - liefert den Wert von S und inkrementiert anschließend den Wert von S
  - *atomare Aktion*, d.h. auch bei "gleichzeitigen" Aufruf von mehreren Prozessen erhält jeder Prozess einen eigenen (eindeutigen) Wert

Sequencer - Operationen

- `#include <sequev.h>`
- Anlegen
  `sequencer_t *create_sequencer(key_t key);`
- Ticket holen
  `long sticket(sequencer_t *sequencer);`
- Löschen
  `int rm_sequencer(sequencer_t *sequencer);`
- Compilieren
  `c89 -… -o prg prg.o -lseqev`

Beliebige Zugriffsfolge mit Sequencer und Eventcounts

```
/* Process 1        */      /*  Process 2 */
long t;                     long t;

e=create_eventcounter(K1); e=create_eventcounter(K1);
s=create_sequencer(K2);    s=create_sequencer(K2);

for(;;)                     for(;;)
{                           {
   t = sticket(s);             t = sticket(s);
   …eawait(e,t)…               …eawait(e,t)…

    critical();                 critical();

   …eadvance(e)…               …eadvance(e)…
}                           }
```


Vergleich Semaphore - Seq/Evc

- P( ) - Operation:

```
t=sticket(seq);
eawait(evc,t);
```

- V( ) - Operation:

```
eadvance(evc);
```


Ressourcenverwaltung

- Wer legt Ressourcen an?
  - Aufrufreihenfolge der Prozesse:
    - fixe Reihenfolge (zB Client-Server Systeme)
    - ***beliebig*** (zB talk)

Ressourcenverwaltung

- Wer löscht Ressourcen?
  - Fehlerfreier Programmverlauf
    - Beim Löschen soll die Synchronisation zwischen den Prozessen sicherstellen, dass kein anderer Prozess als der Löschende mehr auf gemeinsame Ressourcen zugreift!
  - Fehlerfall
    - ***Unsynchronisiertes Aufräumen***:
      Fehlerhafter Prozess löscht Ressourcen
    - Synchronisiertes Aufräumen:
      Eigener Kommunikationskanal nötig (aufwendig!)

Startreihenfolge (1/2)

- Semaphore:

```
/* lege semaphore an */
if ((s=seminit(KEY,PERM,1)) == -1)
{
     /* wenn bereits vorhanden, semgrab */

     if ((s=semgrab(KEY)) == -1)
     {
          /* error! */
          …
```

- Sequencer/Eventcounts:

```
seq=create_sequencer(KEY1);    /* error if seq==-1 */
evc=create_eventcounter(KEY2); /* error if evc==-1 */
```


Startreihenfolge (2/2)

- Shared Memory

```
if ((shmid = shmget(KEY, sizeof(SHMSTRUCT),
                    IPC_CREAT | PERMISSION)) < 0) {

          /* Fehlerbehandlung */
          …
```


Zusammenfassung

- Was sollten Sie heute gelernt haben?
  - Umgang mit Shared Memory
  - Wie synchronisiert man mehrere Prozesse?
  - Den Einsatz der Synchronisationskonstrukte
    - Semaphore und
    - Sequencer und Eventcounts
  - Was ist beim Anlegen und Löschen der Ressourcen zu beachten?