Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Synchronisation und Prozesskommunikation

Wie können parallele und nebenläufige Prozesse synchronisiert werden?

Wie wird Datenkonsistenz in parallelen Programmen gewährleistet?

Synchronisationsobjekte: Mutex, Semaphore, Event, Barrier, Timer, Channel, Monitor

Gruppenkommunikation

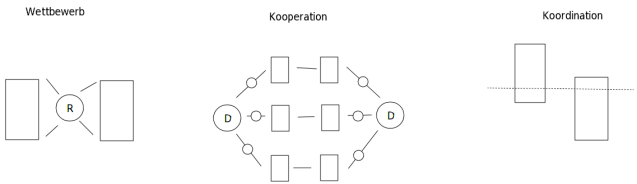
Prozesskommunikation

- Start und Terminierung von Prozessen kann durch die Prozesse selbst, den Prozesskonstruktoren oder explizit durch Synchronisationsobjekte erfolgen
- Aber auch zwischen Start und Terminierung eines Prozesses ist es häufig erforderlich den inneren Prozessfluss von einer Gruppe von Prozessen zu synchronisieren, d.h. an bestimmten Punkten (Milestones) den Kontrollfluss abgleichen.

Synchronisationsklassen

1. Bei geteilten Resourcen und konkurrierenden Zugriff
    - Gegenseitiger Ausshluß und Schutz von kritischen Programmbereichen (Lock)
2. Kooperation
    - Verteilung und Zusammenführung von Prozessen und Daten
3. Koordination
    - Reihenfolge, Abläufe, Aufgabenverteilung

Synchronisationsklassen



Invarianten

> Synchronisationsobjekte müssen bestimmte Bedingungen erfüllen, damit Konsistenz, Sicherheit und Lebendigkeit in parellelen Prozessystemen **jederzeit** gewährleistet sind!

- Eine Invariante ist häufig eine Boolescher oder relationer Ausdruck der sowohl Parameter des Objekts als auch die Prozessinteraktion berücksichtigt

Lock und atomare Operationen

Mutualer Ausschluss mit Lock Objekt

- Ein LOCK Objekt ist ein geteiltes Objekt dass sich in zwei verschiedenen Zuständen σ_{Lock}={*FREE*,*LOCKED*} befinden kann und eine Ressource schützt:

State FREE
Die Ressource ist nicht in Verwendung

State LOCKED
Die Ressource wird bereits von einem Prozess verwendet, andere Prozesse müssen warten

Lock und atomare Operationen

- Ein LOCK Objekt stellt Prozessen zwei Operationen zur Verfügung:

Operation acquire (lock)

Ein Prozess kann eine Ressource anfordern. Ist der Lock im Zustand σ_{Lock}=*FREE*, wird dem anfordernden Prozess die Ressource gewährt (Zustandsübergang σ_{Lock}: *FREE* ⇒ *LOCKED*), andernfalls wird er blockiert bis die Ressource (vom Eigentümer) freigegeben wird.

Lock und atomare Operationen

Operation release (unlock)

Ein Prozess gibt eine zuvor mit *acquire* beanspruchte Ressource wieder frei (Zustandsübergang σ_{Lock}: *LOCKED* ⇒ *FREE*).

Ein Prozess muss die Operationen *acquire* und *release* immer paarweise verwenden!

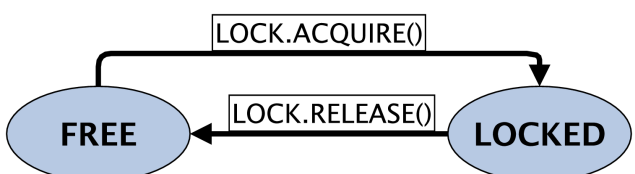


Abb. 1. Zustandsübergangsdiagramm des Lock Objekts

Lock und atomare Operationen

- Beispiel der Nutzung eines Lock Objekts für den Schutz von kritischen Bereichen beim Zugriff auf geteilte Ressourcen

```
OBJECT L: LOCK
VAR V
PAR
  PROCESS P1 IS                          PROCESS P2 IS
  WHILE (True)                           WHILE (True)
    non-critical section                   non-critical section
    L.ACQUIRE()                            L.ACQUIRE()
     critical section(V)                    critical section(V)
    L.RELEASE()                            L.RELEASE()
    non-critical section                   non-critical section
  DONE                                   DONE
```

- Das Lock Objekt löst das mutuale Ausschlussproblem (mutual exclusion), und wird daher auch als Mutex bezeichnet.
- Mehr als zwei Prozesse können mit einem Lock synchronisiert werden.

Lock und atomare Operationen

Synchronisation durch Blockierung

- Ereignisbasierte Synchronisation blockiert die Ausführung von Prozessen bis das Ereignis auf das gewartet wurde eingetreten ist.
  - Hier die Freigabe eines belegten Lock Objekts

Atomare Ressourcen: Register

- Ein Schreib-Lese Register ist die einfachste Form einer geteilten Ressource und wird für die Implementierung eine Lock Objektes benötigt.
  - Aber: Operationen auf dieses Register müssen atomar und konfliktfrei sein, d.h., sequenzialisierbar sein!

> Der Zugriff auf ein atomares Register *R* erfolgt mittels zweier Operationen *op*={*read*,*write*}: *R.read* (), welches den aktuellen Wert von *R* zurück liefert, und *R.write* (v), welche einen neuen Wert in das Register schreibt. Dabei ist ein Konflikt durch konkurrierende Schreib- und Lesezugriffe von einer Menge von Prozessen durch mutualen Ausschluss aufgelöst!

Def. 1. Atomares Register

Eigenschaften eines atomaren Registers

Ein atomares geteiltes Register erfüllt dabei folgende Eigenschaften:

- Zugriffe (*op*=*read*/*write*) erscheinen als wären sie zu einem einzigen Zeitpunkt τ(*op*) ausgeführt worden.

- Der Zeitpunkt der Ausführung der Operation liegt garantiert innerhalb des Intervalls:
  - $\tau(op_S) \leq \tau(op) \leq \tau(op_E)$, mit
  - S: Start und E: Ende der Operation (gemeint ist: Aktivierung durch Prozess)
- Für zwei verschiedene Operationen op_1 und op_2 gilt:
  - $\tau(op_1) \neq \tau(op_2)$

Lock und atomare Operationen

- Jede Lese-Operation gibt den Wert zurück der von der am nächsten (in der Vergangenheit) liegenden Schreibe-Operation resultiert.

> Das bedeutet: ein atomares Register verhält sich so als ob die Ausführung aller Operationen sequenziell ausgeführt worden wäre.

Komposition mit atomaren Objekten

- Atomarität erlaubt die einfache Komposition von geteilten (atomaren) Objekten zu neuen geteilten und atomaren Objekten ohne zusätzlichen (Synchronisations-) Aufwand.
- D.h. wenn eine *Menge von für sich atomaren Registern* $\{R_1,R_2,..\}$ mit den Operationen $\{R_1.read,R_1.write\}$, $\{R_2.read,R_3.write\}$, .. zu einem neuen Objekt $(R_1,R_2,...)$ zusammengefasst (komponiert) werden, so ist dieses ebenfalls *atomar*.

Lock und atomare Operationen { - }

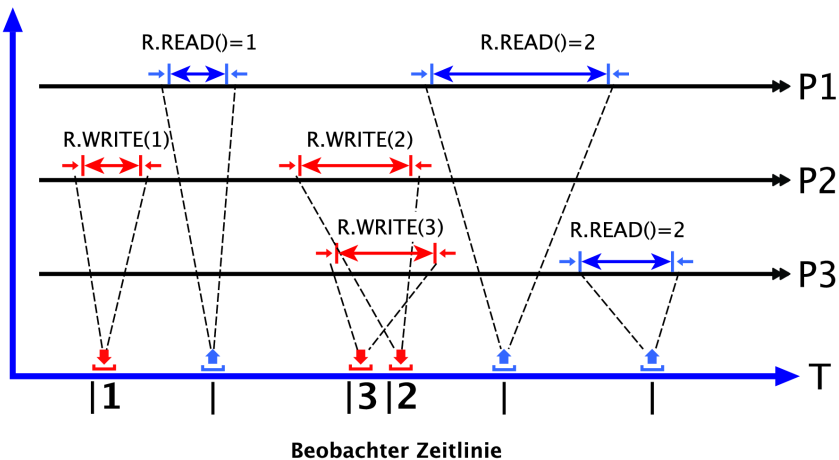


Abb. 2. Beispiel von konkurrierenden und zeitlich überlappenden Schreib/Lese Zugriffen auf ein atomares Register

Mutex: Der Mutuale Ausschluß

Protokolle

- Es muss in jedem Prozess der einen kritischen Bereich ausführen muss ein Eingangs- und Ausgangsprotokoll eingehalten werden damit der mutuale Ausschluss garantiert werden kann.
- Für die Implementierung dieses Protokolls soll gelten:

1. Mutualer Ausschluss und Einhaltung der Lock Invariante (logische Bedingung): Höchstens ein Prozess zur Zeit führt seinen kritischen Abschnitt aus.
2. Keine Deadlocks und Livelocks: Wenn zwei oder mehr Prozesse versuchen, ihre kritischen Abschnitte zu betreten, wird mindestens einer erfolgreich sein.
3. Niedrige Latenz (niedrige Verzögerung): Wenn ein Prozess versucht, in seinen kritischen Abschnitt einzutreten, und die anderen Prozesse ihre nicht kritischen Abschnitte ausführen oder beendet haben, wird der erste Prozess nicht daran gehindert, in seinen kritischen Abschnitt einzutreten.
4. Garantierter Eintritt: Ein Prozess, der versucht, in seinen kritischen Abschnitt einzutreten, wird schließlich erfolgreich sein.

Mutex Objekt

Eine Mutex ist das einfachste aber wichtigste Synchronisationsobjekt für parallele Systeme. **Es dient der Datenkonsistenz durch Schutz von krtischen nicht atomaren Ausführungsbereichen**.

Das Objekt bietet mindest zwei Operationen:

lock (acquire)
: Ein Prozess versucht den Mutex Lock zu erlangen. Wenn dieser Prozess der einzige Bewerber ist bekommt er den Lock und kann in der Ausführung fortfahren. Ist der Lock bereits an einen anderen Prozess vergeben, wir der Aufrufer blockiert (Warten).

unlock (release)
: Ein Prozess gibt den Lock zurück. Wenn es wartende Prozesse gibt wird maximal einer ausgewählt und erhält den Lock.

Mutex Objekt

Lua CSP

```
local mu = Mutex()

Par({
  function () .. mu:lock() .. mu:unlock() .. end,
  function () .. mu:lock() .. mu:unlock() .. end,
  function () .. mu:lock() .. mu:unlock() .. end,
  ...
}, {
  mu = mu
})
```

Beispiele für den Einsatz einer Mutex

- Nicht atomare Operationen auf geteilte Dateneregister

```
local mu = Mutex()
local x  = Register:new('float');
Par({
  function ()
    mu:lock(); x:write(x:read() + 1); mu:unlock()
  end,
  function ()
    mu:lock(); x:write(x:read() - 1); mu:unlock()
  end,
}, {
  mu = mu,
  x = x
})
print(x:read())
```

- Nicht atomare Operationen auf geteilte Datenestrukturen, z.B. Listen oder Graphen

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT Mu

Mutex

Invariante

> Zu jedem Zeitpunkt *t* darf maximal ein Prozess den Lock besitzen. Weitere Anfragen des Locks müssen garantiert abgewiesen werden. Nur ein Prozess darf maximal den Wettbewerb gewinnen.

- Man unterscheidet schwache und starke Mutex Objekte:
  - Bei starken Mutex Objekten darf nur der Prozess den Lock freigeben der ihn agefordert hat und besitzt. Ein Versuch eines Nichteigentümers führt zu einem Programmfehler (Versagen)
  - Bei schwachen Mutex Objekten darf jeder Prozess den Lock freigeben (und auch wenn dieser garnicht vergeben ist)

Semaphore

Eigenschaften

- Ein Semaphore Objekt *S* ist ein Lock basiertes Synchronisationsobjekt und ein **geteilter Zähler mit Warteschlange** *S.counter*, das folgende Eigenschaften erfüllt:
  - Die Bedingung $S.counter \geq 0$ ist immer erfüllt.
  - Der Semaphorenzähler wird mit einem nicht-negativen Wert s_0 initialisiert.
  - Es gibt zwei wesentliche Operationen {`S.down, S.up`} um den konkurrierenden und synchronisierenden Zugriff auf eine Semaphore zu ermöglichen.

Semaphore

- Wenn *#(S.DOWN)* und *#(S.UP)* die Anzahl der Operationsaufrufe ist, die terminiert sind, so gilt die Invariante:

$$S.counter = s_0 + \sum(S.up) - \sum(S.down)$$

Semaphore

Operationale Semantik

Operation down (wait)

S.down erniedrigt den Zähler *S.counter* atomar um den Wert 1 sofern *S.counter > 0*.
Wenn *S.counter=1* und mehrere Prozesse versuchen gleichzeitig die *down* Operation anzuwenden, so wird einer den Wettbewerb gewinnen (Mutualer Ausschluss = Lock). Wenn *S.counter = 0* dann werden Prozesse blockiert und in einer Warteschlange *S.Q* eingereiht.

Operation up (signal)

S.up erhöht den Zähler *S.counter* atomar um den Wert 1. Diese Operation kann immer ausgeführt werden, und blockiert nicht die Ausführung des aufrufenden Prozesses.
Wenn *S.counter = 0* ist und es blockierte Prozesse gibt, wird einer aus *S.Q* ausgewählt und freigegeben (d. h. der Zähler ändert sich effektiv nicht).

Semaphore

Eine Semaphore mit s_0=1 ist eine **binäre Semaphore** vergleichbar mit einer **Mutex**! Eine Semaphore ist stark wenn die blockierten (wartenden) Prozesse in der Reihenfolge wieder freigegeben werden in der sie blockiert wurden (First In First Out / FIFO Ordnung).

Lua

```
local sem = @Semaphore(init)
Par({
  function () P:down() .. C:up()end,
  function () P:down() .. C:up()end,
  function () loop(2) P:up() .. loop(2) C:down() end
},{ P = Semaphore(0), C =Semaphore(0) })
```

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT SemaProCon

Algorithmus

```
OBJECT SEMAPHORE(init) IS
  VAR counter:INT
  OBJECT Q:QUEUE, L:LOCK
  OPERATION down(i) IS
    L.lock()
    IF counter = 0 THEN
      Q := Q + {i}
      |L.unlock(),BLOCK(i)|
    ELSE
      counter := counter - 1
      L.unlock()
    END IF
    RETURN
  END OPERATION
```

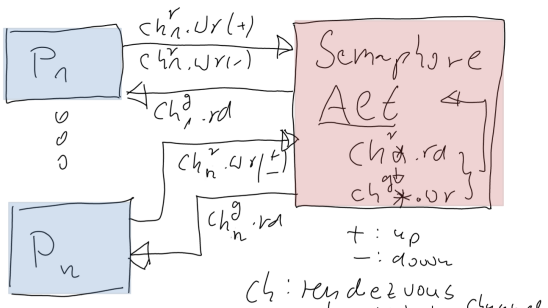
```
OPERATION up(i) IS
  L.lock()
  IF counter = 0 THEN
    IF Q = {} THEN
      counter := counter + 1
    ELSE
      j := Head(Q), Q := Tail(Q)
      UNBLOCK(j)
    END IF
  ELSE counter := counter + 1
  END IF
  L.unlock()
  RETURN
END OPERATION
```

Alg. 1. Semaphore mit Lock Objekt

Semaphore

Implementierung mit Channels

- Verwendung von voll synchronisierten Channels (Rendezvous):
  - Vom Teilnehmer- zum Semaphoreprozess ein Request Channel ch^r
  - Vom Semaphore- zum Teilnehmerprozess ein Ack. Channel ch^g
  - Semaphoreprozess benutzt *Alt* Operator für alle Request Channels



Semaphore - Produzenten-Konsumenten Systeme

- Semaphoren können eingesetzt werden um konkurrierende Produzenten-Konsumenten Probleme zu lösen. Beispiel ist ein Pufferspeicher mit First-In First-Out Reihenfolge, der eine endliche Anzahl von Speicherzellen eines geschützten Speichers *Buf*[0..*k*-1] mit *k*=*size* besitzt, und die beiden Zustände *S={FREE, FULL}* annehmen kann.
- Schutz des Speichers bedeutet: gleichzeitiger Zugriff zweier Prozesse auf gleiche Speicherzelle *Buf*[*x*] muss verhindert werden.
- Schutz durch zwei Semaphoren *FREE* (*k*) und *BUSY* (0).

Implementierung eines Pufferspeichers mit Semaphoren

```
OBJECT BUFFER (size) IS
  VAR BUF: datatype[size], in,out: INT
  OBJECT FREE: SEMA(size),
         BUSY:SEMA(0)
  OPEARTION produce(v) IS
    FREE.down()
    BUF[in].write(v)
    in := (in+1) mod size
    BUSY.up()
  END OPERATION
  OPERATION consume IS
    BUSY.down()
    r := BUF[out].read()
    out := (out+1) mod size
    FREE.up()
    RETURN r
  END OPERATION
END OBJECT
```

Alg. 2. Synchronisierter Pufferspeicher (Queue)

Semaphore - Produzenten-Konsumenten Systeme

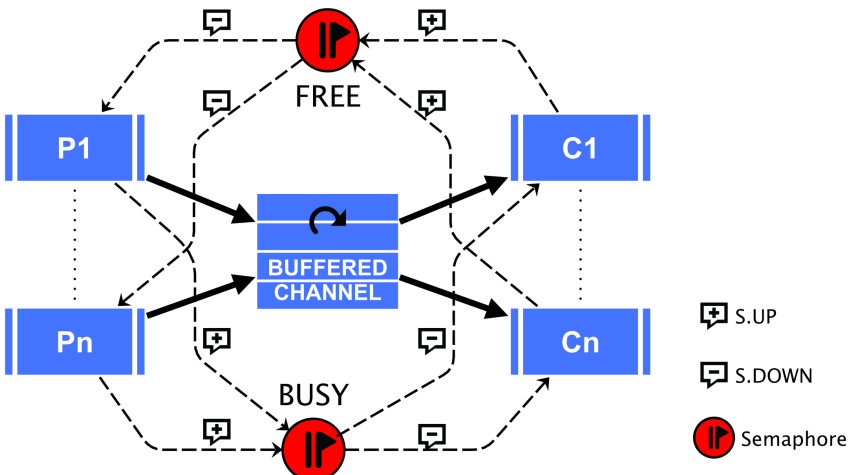
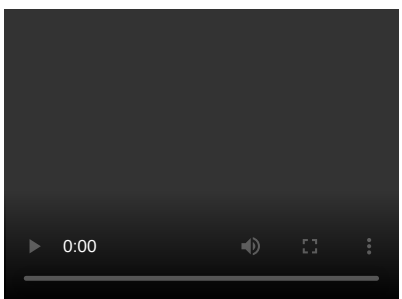


Abb. 3. Synchronisation von Produzenten und Konsumenten von Daten mittels zweier Semaphoren

Semaphore - Dining Philosophers

- Das Problem der fünf dinierenden Philosophen ist ein Beispiel für Deadlocks in verteilten und parallelen (asynchronen) Systemen mit dem Communicating Sequential Processes Modell!



- 5 Philosophen an einem Tisch
- 5 Gabeln, jeweils eine zwischen zwei Philosophen
- Ein Philosoph braucht zwei Gabeln zum Essen!
- Eine Gabel wird durch eine Semaphore repräsentiert (atomare Ressource, Startwert des Zählers ist 1)

Semaphore - Dining Philosophers

```
OBJECT ARRAY Forks [N]: SEMA(1)
PROCESS ARRAY Philosopher [N]
  FOR try = 1 TO 10 DO
    THINKING ...
    Forks[#id].down()
    Forks[#id+1 % N].down()
    EATING ...
    Forks[#id+1 % N].up()
    Forks[#id].up()
  DONE
END
```

Code 1. Dinining Philosophers Template

Semaphore - Dining Philosophers

- Klassisches Problems das zu Deadlocks führt!

Deadlockefreie Lösungen

1. Einführung einer zentralen Instanz die die Gabeln mittels Prioritäswarteschlangen verwaltet
   - Prozess fragt mit einer Nachricht zwei Gabeln an (Input Channel)
   - Verwalter prüft Verfügbarkeit und gewährt die Gabeln durch eine Antwortnachricht (Output Channel)
   - Ein Prozess gibt Gabeln mit einer Nachricht wieder zurück

Semaphore - Dining Philosophers

2. Abfrage des Semaphorenwertes beider Gabeln bevor die Ressource angefragt wird ggf. über eine globale Mutex geschützt:

```
sema[left]:down();
if sema[right]:level() == 1 then sema[right]:down()
else sema[left]:up(); sleep(random time) end
```

3. Münzwurf und Reihenfolge des Zugriffs: Eine Zufallszahl zwischen 0..1 entscheidet ob zuerst die linke und dann rechte Gabel beansprucht wird oder umgekehrt.

Ergeinisobjekt (Event)

- Ein Event ist ein Signalobjekt. Ein oder mehrere Prozesse können auf das Eventsignal warten. Ein weiterer Prozess signalisiert schließlich das Event

Es gibt zwei Operationen:

Opeartion await
: Ein Prozess wartet auf das Eintreten des Ereignisses (das Signal). Der Prozess blockiert bis das Signal eintrifft und wird in eine Warteschlange Q eingereiht.

Operation wakup
: Ein Prozess signalisiert das Ereignis. Alle in die Warteschlange befindlichen Prozesse werden aktiviert.

Lua CSP

```
local ev = Event()
Par({
  function () ev:await() .. end,
  function () ev:await() .. end,
  function () ev:wakeup() .. end
},{ ev:ev })
```

Wenn das Signal vor dem Erwarten ausgelöst wird ist das Signal verloren und wartende Prozesse werden nicht frei gegeben!

Barriere

- Um das Problem des Wettrennens bei Events zu vermeiden ist es besser eine Barriere für eine Prozessgruppe bekannter Größe zu verwenden.
- Da Barriere ist selbstauslösend, d.h. der letzte Prozess löst das Signal aus.

```
local ba = Barrier(groupsize)
Par({
  function () ba:await() .. end,
  function () ba:await() .. end,
  function () ba:await() .. end
},{ ba:Barrier(3) })
```

Barriere

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT Barrier

Timer

- Ein Timer ist ein automatisch synchronisierendes Event

```
local t = Timer(interval, once, fiber?)
Par({
  function () loop t:await() .. end
  function () loop t:await() .. end
  function () t:start() .. end
},{t:Timer(1000,false)}}
```

Def. 2. Lua CSP Timer

Channel

- Ein Kanal ist ein synchronisierter Pufferspeicher mit First-In First-Out Reihenfolge und wird in Produzenten-Konsumenten Systemen eingesetzt.
- Operationen und Verhalten

Operation read

Ein Datenelement wird aus dem Puffer gelesen. Wenn der Puffer leer ist wird der Prozess in eine Wartschlange eingereiht: $Q=Q \cap \{P\}$

Operation write

Ein Datenelement wird in den Puffer geschrieben. Wenn die Warteschlange $Q \neq \emptyset$, dann wird ein Prozess entfernt und die ausstehende *read* Operation ausgeführt.

Ist die Puffergröße Null, dann Rendezvous Handshake!

Channel

```
local ch = Channel(size,fiber?)
Par({
  function () local v = ch:read() .. end,
  function () ch:write(ε) .. end
},{ch=ch})
```

Def. 3. Lua CSP Channels

Channel

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT ChannelProCon

Der Alt Prozesskonstruktor

- Die Arbeit mit Channels kann bedeuten dass ein Prozess aus mehreren Channels gleichzeitig Daten lesen muss, aber aufrgund des synchronen Verhaltens nicht alle Channel einer Gruppe $Ch=\{ch_i\}$ abfragen kann.

- Ein Lösung bietet der Alt(ernative) Prozesskonstruktur der es ermöglich auf das Eintreten mehrerer Ereignsse (hier die Verfügbarkeit von Daten in Channels) zu warten und **eines auszuwählen**.

- Der Alt Prozess garantiert Mutual Exclusion Verhalten, d.h.:

  - Treten mehrere Ereignisse gleichzeitig ein, wird genau eines (wie auch immer) ausgewählt und ein Verarbeitungsprozess gestartet
  - Nur maximal eine Operation (also *channel*:*read*) wird aus $Ch=\{ch_i\}$ ausgeführt, auch wenn die anderen bereist eingetretende Ereignisse (Daten) haben (Mikro Event loop!!)

Alt Prozesskonstruktor

```
ch1 = Channel()
ch2 = Channel() ...
  local x
  Alt({
    function () x=ch1:read(); <optional handler code> end,
    function () x=ch2:read(); <optional handler code> end,
    ..
  })
  <process x>
```

Def. 4. Lua CSP Auswahlprozesskonstruktor. Es wird immer nur einer der Alt Prozesse aktiviert wenn die blockierende Anweisung (read) bereit wird

Monitor

- Mutex und Semaphore sind low-level Synchronisationsobjekte
- Monitore erlauben die Definition von konkurrierenden Objekten auf der Abstraktionsebene der imperativen Programmiersprachen → high-level.
- **Ein Monitor ist eine Generalisierung eines Objekts, welches Daten und Operationen kapselt. Zugriff auf die interne Repräsentation erfolgt durch die Operationen mutual exklusiv.**
    - Aber immer mit dem Ziel die Datenkonsistenz möglichst frein granular zu gewährleisten.

Monitor

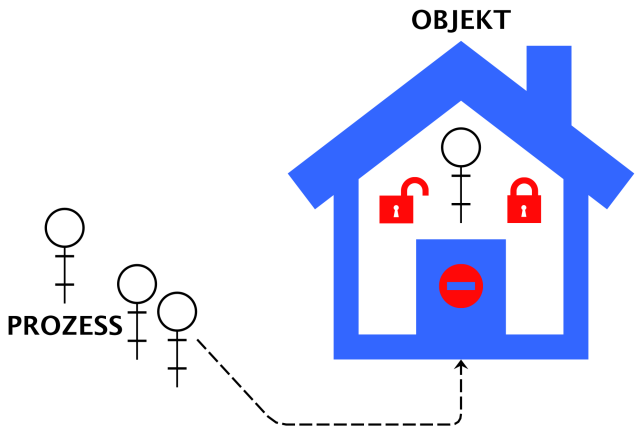


Abb. 4. Objektmonitor mit automatischen Schutz (Datenkonsistenz des Objekts)

Monitor

Konzepte

Mutualer Ausschluss

Der Monitor stellt sicher dass immer nur ein Prozess eine Operation des Objektes ausführen kann ⇒ geschützter kritischer Ausführungsbereich = mutualer Ausschluss.

Ein Prozess muss das Schloss des Monitors öffnen. Nachdem der Prozess den Monitor "betritt" (nutzt), schließt das Schloss automatisch wieder. Wenn der Prozess den Monitor verlässt wird das Schloss wieder automatisch geöffnet.

Monitor

Events: Bedingungsvariablen mit Queues

- Bedingungsvariablen *C* sind Objekte die interne Synchronisation ermöglichen sollen.
- Bedingungsvariablen können nur innerhalb des Monitors durch folgende Operationen verwendet werden:

Monitor

Operation *C.wait* (*cond*)

Diese Operation blockiert = stoppt den aufrufenden Prozess p und reiht ihn in die Warteschlange $C.Q \leftarrow C.Q + \{p\}$ ein. Für den Prozess der nicht mehr aktiv ist, kann der Mutex LOCK aufgehoben werden.

- Die Wait Operation stellte eine boolesche Bedingung *cond* dar auf deren Erfüllung gewartet wird.

Operation *C.signal* ()

Ein Prozess p ruft diese Operation auf und erfüllt damit die Bedingung *cond*.

- Wenn $C.Q = \emptyset$ ist dann hat die Operationen keinen Effekt
- Wenn $C.Q = \{q,...\}$ dann wird ein Prozess q aus Q ausgewählt und aktiviert, so dass $C.Q \leftarrow C.Q \setminus \{q\}$

Monitor

Bewahrung des mutualen Ausschlusses in Fall ii.) erforderlich:

- Der aktivierte Prozess *q* fährt innerhalb des Monitors fort (bekommt den Eintritt wieder) und führt die Anweisungen nach *C.wait* () aus.

Monitor

- Der aktivierende Prozess *p* wird blockiert, hat aber höchste Priorität den Monitor danach wieder zu erlangen um die Anweisungen nach *C.signal* () auszuführen.

Operation *C.empty* ()
Liefert das Ergebnis für die Bedingung $C.Q = \emptyset$

- Algorithmik der Bedingungsvariablen (vereinfacht)

```
OBJECT CONDITION
  (M:MONITOR) IS
  OBJECT q: QUEUE                OPERATION signal IS
     p: PROCESS                IF q != {} THEN
  OPERATION wait IS                  p := Head(q), q := Tail(q)
    p := MYSELF()                    UNBLOCK(p)
    q := q + {p}                   END IF
    |BLOCK(p) & M.l.unlock()|   END OPERATION
  END OPERATION
```

Monitor

- Implementierung einer Semaphore mit Bedingungsvariablen (vereinfacht)

```
MONITOR SEM IS
  VAR s:INT
  OBJECT c: CONDITION
  OPERATION down IS          OPERATION up IS
    IF s=0 THEN c.wait()       s := s + 1
      s := s - 1               c.signal();
    RETURN                     RETURN
  END OPERATION              END OPERATION
```

Alg. 3. Semaphore mit Bedingungsvariablen

Monitor

- Um den mutualen Ausschluss zu garantieren muss Vorrang der Prozessausführung und Aktionen mit Bedingungsvariablen definiert werden.
  - Wartende bereite Prozesse (Menge $\mathbb{W}$) deren Bedingung *cond* erfüllt ist.
  - Signalisierende Prozesse (Menge $\mathbb{S}$)
  - Prozesse die noch keinen Eintritt in den Monitor erhalten haben (blockiert, Menge $\mathbb{E}$)
  - Prioritäten: Prio($\mathbb{W}$) > Prio($\mathbb{S}$) > Prio($\mathbb{E}$)

Monitor

Ein signalisierter (blockierter) Prozess der auf eine Bedingung *C* wartet wird sofort aktiviert (d. h. die Bedingung ist erfüllt).

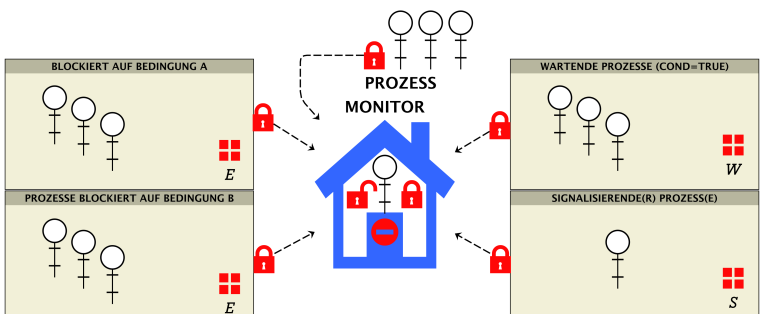


Abb. 5. Verschiedene Prozessmengen bei Monitoren

Monitor

- Würde nach Aktivierung des wartenden Prozesses (*cond*=TRUE) auch der signalisierende Prozess im Monitor aktiv sein könnte er die Bedingung, die der Bedingungsvariablen zu Grunde liegt, wieder ungültig (*cond*=FALSE) machen und müsste ebenfalls wieder blockiert werden.

- Es gilt daher für den Monitor die Erfüllung eines Prädikats *P* (Boolesche Bedingung), das die Datenkonsistenz des Objektes sicher stellen muss.

Varianten

signal-and-wait

Der signalisierende Prozess wartet nach der Signalisierung und der mutuale Ausschluss ist sichergestellt und die Wahrung *P*=TRUE. Der wartende Prozess führt aus: IF ¬*P* THEN *C.wait* () END IF

signal-and-continue (re-check)

Der signalisierende Prozess läuft weiter! Das könnte zur Verletzung des Prädikats *P* führen, daher muss der signalisierte Prozess nochmals die Gültigkeit *P*=TRUE prüfen so dass für ihn auszuführen ist: WHILE ¬*P* DO *C.wait* () END WHILE

Objektmonitore in Lua

```
local S = class()
function S:init (i)
  self.counter=i;  self.L=Mutex()
end
function S:up ()
  self.counter=self.counter+1
end
function S:down ()
  self.counter=self.counter+1
end
function S:__index (index)
  local value = rawget( self, index )
  if value == nil then
    return function (...)
      rawget( self, 'L' ):lock();  -- LOCK
      rawget(getmetatable(self), index)(...) -- CALL PROT
      rawget( self, 'L' ):unlock() -- UNLOCK
    end
  else return value end
end
```

Code 2. Beispiel für einen einfachen und naiven geschützten Monitor in Lua. Wo könnte es Probleme geben?

Zusammenfassung

Es gibt verschiedene Synchronisations- und Kommunikationsobjekete für Prozesssysteme (parallel wie sequenziel) ⇒
Interprozesskommunikation

Häufige Objekte: Mutex, Semaphore, Event, Barriere, Timer, Channel, Monitor

Diese Objekte müssen Invarianten erfüllen die zu keinem Zeitpunkt verletzt werden dürfen (d.h. ungültig werden)