Echtzeitsysteme

3. Synchronisation und Kommunikation: Gemeinsame Daten (*Shared Variables*)

Prof. Dr. Roland Dietrich

Synchronisation und Kommunikation

- Die Korrektheit eines nebenläufigen Programms hängt ab von der Synchronisation und Kommunikation seiner Tasks
- **Synchronisation**
  - Einhalten von Bedingungen/Einschränkungen bzgl. des zeitlichen Reihenfolge von Aktivitäten unterschiedlicher Tasks
    - z.B. Aktion A in Task T muss immer vor Aktion B von Task S stattfinden
- **Kommunikation**
  - Austausch von Daten zwischen Tasks
- Kommunikation setzt Synchronisation voraus
- Synchronisation kann als Kommunikation ohne Inhalt betrachtet werden
- Formen der Kommunikation
  - Kommunikation über gemeinsame Daten (*shared variables*)
  - Kommunikation durch Nachrichten (*message passing*)

    → Kapitel 4

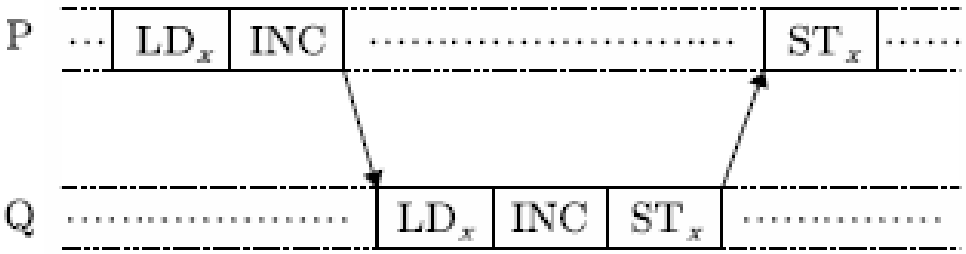
- Formen der Kommunikation
  - Kommunikation über Nachrichten erfordert eine „Infrastruktur“
    - zum Senden, Zwischenspeichern und Empfangen von Nachrichten
  - Kommunikation über gemeinsame Daten ist gut zu realisieren auf einem Multiprozessorsystem mit gemeinsamem Speicher
    - Aber auch eine Nachrichten-Austausch kann dort realisiert werden
  - Kommunikation über Nachrichten ist gut zu realisieren auf einem verteilten System
    - Aber auch gemeinsame Daten können dort unterstützt werden
  - Welche Kommunikationsform realisiert wird, ist eine Entscheidung, die der Sprach- oder Betriebssystem-Designer unabhängig von der Ziel-Architektur treffen kann
  - Aus Sicht der Anwendungsprogrammierung sind beide Formen gleich mächtig (bereits 1978 nachgewiesen!)

Konsistenzproblem

- Beispiel:
  - Zwei nebenläufige Tasks P und Q greifen auf dieselbe Variable X zu.

```
P: ...                    Q: ...
   X = X + 1;                X = X + 1;
   ...                       ...
```

  - Problem: die Inkrementierung von X wird auf dem Prozessor nicht unbedingt „atomar“ (unteilbar) ausgeführt, z.B.
    1. *Lade den Wert von X in den Akkumulator (LD_X)*
    2. *Inkrementiere den Akkumulator (INC)*
    3. *Speichere den Inhalt des Akkumulators nach X (ST_X)*
  - Mögliche Zeitliche Reihenfolge der Ausführung:



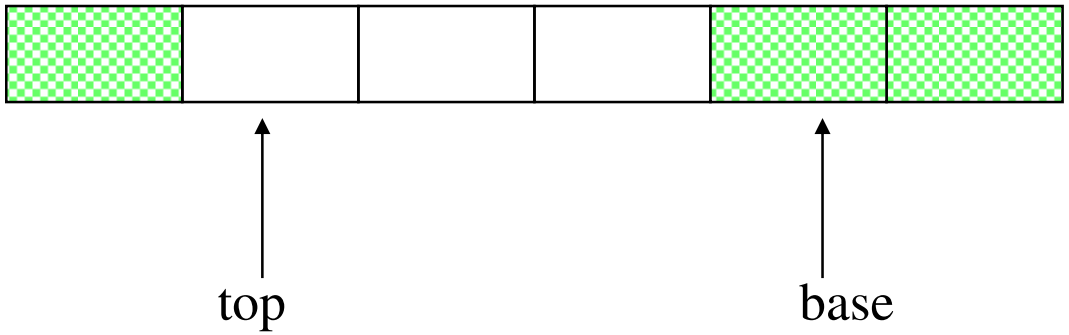
Quelle: [Zöbel 2008, Abb. 2.30]

→ X wird nur einmal inkrementiert!

Konsistenzproblem

- **Kritische Bereiche und gegenseitiger Ausschluss**
  - Eine Folge von Aktionen, deren Ausführung nicht unterbrochen werden darf heißt **kritischer Bereich** (***critical section***)
    - Ein kritischer Bereich muss **atomar** ausgeführt werden
  - Der Schutz von kritischen Bereichen durch entsprechende Synchronisierungsmaßnahmen heißt **gegenseitiger Ausschluss** (***mutual exclusion***)
  - Welche Aktivitäten sind per se atomar?
    - Einzelne Speicherzugriffe (auf „Wort-Ebene“) sind atomar
      - Beispiel: Nebenläufiges Ausführen der Anweisungen X := 5 und X := 6 ergibt 5 oder 6 als Wert von X (es kann sich keine andere Task „dazwischen schieben“)
    - Der Zugriff auf strukturierte Objekte ist nicht atomar!
      - Nur die Manipulation einzelner Daten-Worte der Struktur ist atomar

- **Bedingungs- Synchronisation**
  - Eine Task kann eine Aktion nur dann ausführen, wenn eine andere Task eine bestimmte Bedingung erfüllt
    - z.B. selbst eine bestimmte Aktion ausgeführt hat
    - z.B. einen bestimmten Zustand hergestellt hat
  - Beispiel: Produzenten/Verbraucher-Systeme (*producer/consumer*)
    - Eine Task produziert Daten, eine andere (nebenläufige) Task verbraucht die Daten
    - Synchronisationsbedingungen bei Verwendung eines endlichen, beschränkten Puffers (*bounded buffer*)
      - Die Produzenten-Task darf keine Daten im Puffer ablegen, wenn er voll ist
      - Die Verbraucher-Task darf keine Daten aus dem Puffer entnehmen, wenn er leer ist



Busy Waiting

- Idee:
  - Um zu signalisieren, dass eine Synchronisationsbedingung erfüllt ist, setzt eine Task einen Wert in einem **Flag** („Sperrbitt“, „Schlossvariable“)
  - Um auf das Eintreten der Bedingung zu warten, prüft eine andere Task ständig das Flag (Warteschleife, „busy loop“)

```
task P1;  -- Wartende Task
  ...
  while flag = down do null; end;
  ...
end P1;

task P2;  -- Signalisierende Task
  ...
  -- Synchronisationsbedingung erfüllt
  flag = up;
  ...
end P2;
```

Busy Waiting

- Probleme
  - Ineffizient:
    - Tasks belegen Prozessorzyklen mit unnützem Warten
    - Währenddessen kann nichts Sinnvolles ablaufen
  - Scheduling ist schwierig, wenn sich mehr als eine Task in der Warteschleife befindet
  - „Livelock“ möglich: Tasks bleiben in der Warteschleife hängen
  - Um kritische Bereiche zu schützen, ist weiterer Aufwand erforderlich
    - Naiver Ansatz: *flag := up ←→ Kritischer Bereich freigegeben*

```
task P1;
  ...
  while flag = down do null end;
  -- jetzt ist flag = up
  flag := down;
  <critical section P1>
  flag := up;
  ...
end P1;
```

```
task P2;
  ...
  while flag = down do null end;
  -- jetzt ist flag = up
  flag := down;
  <critical section P2>
  flag := up;
  ...
end P2;
```

Busy Waiting

- Algorithmus von Petersen (1981)
  - Jeder Prozess manipuliert ein „Ankündigungs-Flag“
    - Flag := up ←→ „Ich möchte in einen kritischen Bereich eintreten“
    - Flag := down ←→ „Ich bin fertig mit dem kritischen Bereich“
  - Wer „dran“ ist, regelt ein zweites Flag (***turn***)
  - Jede Task räumt der anderen zunächst den Vortritt ein
  - Gegenseitiger Ausschluss ist sicher, livelock unmöglich.

```
Task P1;
  loop
    flag1 := up;
    turn := 2;
    while flag2 = up
          and turn = 2
    do null; end;
    <critical section>
    flag1 = down;
    <non-critical section>
  end
end P1;
```

```
Task P2;
  loop
    flag2 := up;
    turn := 1;
    while flag1 = up
          and turn = 1
    do null; end;
    <critical section>
    flag2 = down;
    <non-critical section>
  end
end P1;
```

Busy Waiting

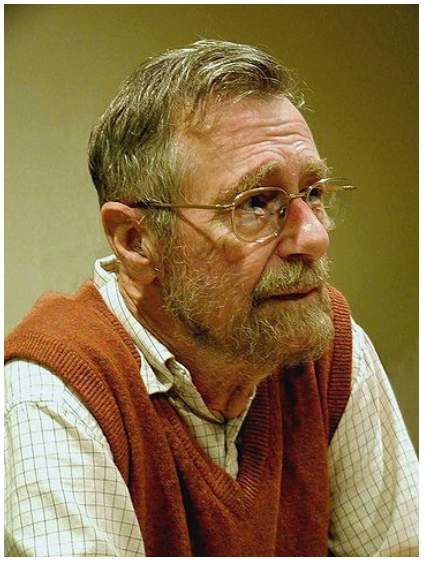
- Zuverlässigkeit von Petersens Algorithmus
  - Jeder kritische Bereich kommt dran („*fairness*“, kein *livelock*)
  - Fehlerzustände im Nicht-Kritischen Bereich beeinflussen nicht die andere Task
  - Fehlerzustände im kritischen Bereich können aber zu *livelock* führen

- Fazit: Probleme mit *Busy Waiting*
  - Schwierig zu entwerfen und zu verstehen
  - Schwierig als korrekt zu Beweisen
    - Aufgabe: Verallgemeinerung von Petersens Algorithmus auf n Tasks!
  - Ineffizient (Prozessor wird mit Warteschleifen beschäftigt)
  - Eine „brutale“ Task, die den gemeinsamen Speicher „missbraucht“, d.h. sich nicht an das Protokoll hält, kann die ganze Steuerung zunichte machen

  → In nebenläufigen Programmiersprachen werden höhere Konzepte zur Synchronisation verwendet (z.B. Semaphore und Monitore)

Semaphore

- Ein **Semaphor** ist eine *nicht-negative Integer-Variable* **S** die außer der Initialisierung nur durch zwei Operationen ***Wait()*** und ***Signal()*** bearbeitet werden kann:
  - **Wait(S):** **if** S > 0
    then S := S-1
    else begin
    Task verzögern bis S > 0;
    S := S-1
    end;
  - **Signal(S):** **S := S + 1;**
- *Wait()* und *Signal()* müssen als **atomare Operationen** realisiert werden.
  - *Anmerkungen:*
    - Die Operationen *wait* und *signal* werden häufig auch mit *P* und *V* bezeichnet.
    - „Task verzögern“ bedeutet, dass sie den Prozessor nicht mehr belegt



Quelle: Wikipedia,
© 2002 Hamilton Richards

Edsger W. Dijkstra
1930 – 2002

Turing Award 1972

Pionier im Bereich Algorithmen (Kürzeste Wege) und strukturiertem Programmieren

Semaphore

- Bedingungs-Synchronisation mit Semaphoren

```
consyn : semaphore = 0; -- Inialisierung mit 0

task P1; -- Wartende Task
   ...
   wait(consyn);
   ...
end P1;

task P2; -- Signalisierende Task
   ...
   -- Synchronisationsbedingung erfüllt
   signal(consyn);
   ...
end P2;
```

Semaphor-Variable = 1 bedeutet, dass die Synchronisationsbedingung erfüllt ist.

Semaphore

- Gegenseitiger Ausschluss mit Semaphoren

```
mutex: semaphore = 1; -- Initialisierung mit 1

task P1;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P1;

task P2 ;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P2;
```

Mit dem **Initialwert** für die Semaphor-Variable kann festgelegt werden, wie viele Tasks sich gleichzeitig in einem kritischen Bereich aufhalten dürfen.

Der **aktuelle Wert** sagt, wie viele Tasks den kritischen Bereich noch betreten können.

Semaphore

- Blockieren von Tasks
  - *wait(S)* mit S=0 führt zu einer „Verzögerung“ der Tasks
  - Möglichkeiten, tasks zu verzögern:
    - Busy Waiting (schlecht!)
    - Entfernen der Task aus der Menge der ausführbaren (executable) Tasks
      → dann belegen Sie während des Wartens keine Ressourcen im Prozessor
  - Task-Zustände bei Verwendung von Semaphoren:
    - ***executable*** *Tasks*: sind **ausführbereit**, und werden ausgeführt, sobald die Taskverwaltung den Prozessor zuteilt
      - werden in der Task-Warteschlange des Prozessors verwaltet
    - ***suspended*** *Tasks*: sind **blockiert**, aber nicht ausführbereit, warten auf die Freigabe
      - werden in einer Semaphor-Warteschlange eingereiht
  - Es ist Aufgabe des Laufzeitsystems (RTSS), die erforderlichen Task-Wechsel-Operationen zu unterstützen

Semaphore

- Implementierung von Wait/Signal
  - Jede Semaphore-Variable S hat eine eigene Warteschlage von blockierten Tasks
  - `number_suspended(S)` ist die Anzahl der Tasks in dieser Warteschlange

```
wait(S):
  if S > 0 then S := S - 1
  else
    number_suspended(S) := number_suspended(S) + 1;
    suspend_calling_task(S);

signal(S):
  if number_suspended(S) > 0 then
    number_suspended(S) = number_suspended(S) - 1;
    make_one_suspended_task_executable_again(S);
  else
    S = S+1;
```

Semaphore

- Verklemmungen (*deadlocks*)
  - Eine Menge von Tasks, die alle blockiert sind und keiner mehr ausführbar werden kann
  - *Beispiel:*

Kein Deadlock!

```
P1:                    P1:
  wait(S1);              wait(S1);
  wait(S2);              wait(S2);
  .                      .
  .                      .
  .                      .
  signal(S2);            signal(S2);
  signal(S1);            signal(S1);
```

Deadlock möglich!

```
P1:                    P1:
  wait(S1);              wait(S2);
  wait(S2);              wait(S1);
  .                      .
  .                      .
  .                      .
  signal(S2);            signal(S1);
  signal(S1);            signal(S2);
```

- Wenn sich nebenläufige Tasks über mehrere Semaphore synchronisieren, sollten die wait/signal-Aufrufe in beiden Tasks in der selben Reihenfolge erfolgen.

Semaphore in Ada

- *Beispiel:* Ein Package für Semaphore in Ada
  [Burns & Wellings 2009, Kap. 5.4.5]
  - Kein Bestandteil der Sprache Ada
  - Kann mit Hilfe anderer Ada-Sprachmittel implementiert werden

```
package Semaphore_Package is
  type Semaphore(Initial: Natural := 1)
       is limited private;
  procedure wait(S: in out Semaphore);
  procedure signal(S: in out Semaphore);
private
  type Semaphore is ...
end Semaphore_Package;
```

Semaphore in Ada

- Erzeuger-/Verbraucher-Problems in Ada (1) vgl. S. 3-6
  - Zwei Nebenläufige Tasks:
    - *Consumer*
    - *Producer*

```
procedure Main is
  package Buffer is
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
  end Buffer;
  task Producer;
  task Consumer;

  package body Buffer is separate;
  -- see next page
  use Buffer;
```

Erforderliche Schutzmaßnahmen:

- Kein gleichzeitiger Zugriff von Consumer und Producer auf den Puffer
- Kein Einfügen in den vollen Puffer
- Keine Entnahme aus dem leeren Puffer

```
task body Producer is
    Item : Integer;
  begin
    loop
      -- produce item
      Append (Item);
    end loop;
  end Producer;

  task body Consumer is
    Item : Integer;
  begin
    loop
      Take (Item);
      -- consume item
    end loop;
  end Consumer;
begin
  null;
end Main;
```

Semaphore in Ada

- Erzeuger-/Verbraucher-Problem in Ada (2)
  - Verwendung eines endlichen Puffers der Größe N
  - Implementierung der Schutzmaßnahmen durch drei Semaphore
    - **Mutex:** Gegenseitiger Ausschluss von Producer und Consumer
    - **Item_Available:** Bedingungssynchronisation „Puffer nicht leer“ (Default=0)
    - **Space_Available:** Bedingungssynchronisation „Puffer nicht voll“ (Default=N)

```
with Semaphore_Package; use Semaphore_Package;
separate (Main)
package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;

  Mutex : Semaphore; -- default is 1
  Item_Available : Semaphore(0);
  Space_Available : Semaphore(Initial => Size);
```

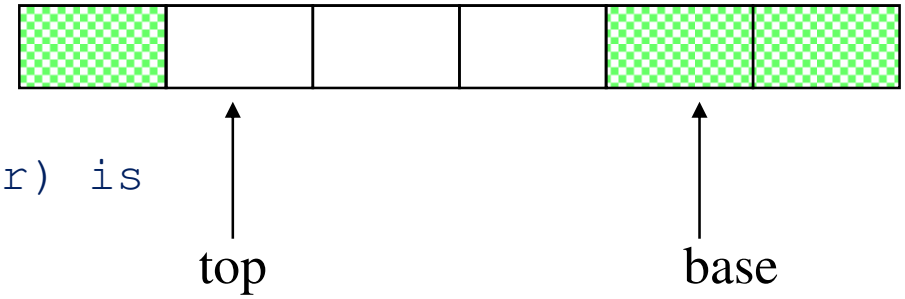
Semaphore in Ada

- Lösung des Erzeuger-/Verbraucher-Problems in Ada (3)

```
procedure Append (I : Integer) is
  begin
    Wait(Space_Available);
    Wait(Mutex);
      Buf(Top) := I; Top := Top + 1;
    Signal(Mutex);
    Signal(Item_Available);
  end Append;

procedure Take (I : out Integer) is
  begin
    Wait(Item_Available);
    Wait(Mutex);
      I := Buf(Base); Base := Base + 1;
    Signal(Mutex);
    Signal(Space_Available);
  end Take;

end  Buffer;
```



Semaphore in C/Real-Time POSIX

- C/Real-Time POSIX-Schnittstelle für Semaphore: `<semaphore.h>`
  - Datentypen:
    - **`sem_t:`** repräsentiert eine Semaphore-Variable
  - Funktionen (u.a.):
    - **`sem_init():`** Initialisieren einer Semaphore-Variable
    - **`sem_wait():`** Standard-*wait*-Operation
    - **`sem_post():`** Standard-*signal*-Operation
    - **`sem_getvalue():`** Liefert den aktuellen Wert einer Semaphore-Variable
  - Genaue Spezifikationen und weitere Funktionen: siehe [Beispiele zu Kapitel 3].
    - Speziell: Bei der Initialisierung eines Semaphors kann man wählen, ob Prozesse (laufen in verschiedenen Adressräumen) oder Threads (laufen in ein und demselben Adressraum) synchronisiert werden sollen.

Semaphore in C/Real-Time POSIX

- Beispiel: Nebenläufiger Zugriff auf Gemeinsame Ressourcen
  - Mehrere Prozesse müssen auf eine gemeinsame Ressource (z.B. einen Drucker) zugreifen.
  - **`allcoate()`**
    - Ein Prozess „beantragt“ den Zugriff auf die Ressource. Falls die Ressource gerade belegt ist, wird der aufrufende Prozess blockiert.
  - **`deallocate()`**
    - Ein Prozess, der die Ressource fertig benutzt hat, gibt sie wieder frei. Dadurch wird der nächste auf die Ressource wartende und blockierte Prozess wieder aktiviert.
  - Prioritäten
    - Es gibt drei Prioritätsstufen (hoch, mittel, niedrig)
    - Nach einem `deallocate()` werden blockierte Prozesse in der Reihenfolge ihrer Prioritäten (hohe zuerst) wieder aktiviert.
  - Konsistenzmaßnahmen:
    - allocate/deallocate müssen sich gegenseitig ausschließen
    - Solange die Ressource belegt („*busy*“) ist, müssen anfordernde Prozesse blockiert werden (Bedingungssynchronisation)

Semaphore in C/Real-Time POSIX

- Beispiel: Gemeinsame Ressourcen (1)

```
#include <semaphore.h>

typedef enum {high, medium, low} priority_t;
typedef enum {false, true} boolean;

sem_t mutex;    /* used for mutual exclusive
                   access to waiting and busy */
sem_t cond[3]; /* used for condition synchronization for each priority */
int waiting;    /* count of number of threads
                   waiting at a priority level */
int busy; /* indicates whether the resource is in use */

void allocate(priority_t P)
{
 SEM_WAIT(&mutex); /* lock mutex */
  if(busy) {
    SEM_POST(&mutex);    /* release mutex */
    SEM_WAIT(&cond[P]); /* wait at correct priority level */
    /* resource has been allocated */
  }
  busy = true;
  SEM_POST(&mutex); /* release mutex */
}
```

- Beispiel: Gemeinsame Ressourcen (2)

```
int deallocate(priority_t P)
{
  SEM_WAIT(&mutex); /* lock mutex */
  if(busy)
  {  busy = false;
     /* release highest priority waiting thread */
     SEM_GETVALUE(&cond[high],&waiting);
     if ( waiting < 0) SEM_POST(&cond[high]);
     else { SEM_GETVALUE(&cond[medium],&waiting);
            if (waiting < 0) SEM_POST(&cond[medium]);
            else { SEM_GETVALUE(&cond[low],&waiting);
                   if (waiting < 0) SEM_POST(&cond[low]);
                   else SEM_POST(&mutex); /* no one waiting, release lock */
                 }
          }
        /* resource and lock passed on to */
        /* highest priority waiting thread */
    return 0;
  }
  else return -1; /* error return */
}
```

Semaphore in C/Real-Time POSIX

- Beispiel: Gemeinsame Ressourcen (3)
  - Initialisierung der Semaphore:

```
void initialise() {
  priority_t i;

  busy = false;
  SEM_INIT(&mutex, 0, 1);
  for (i = high; i<= low; i++) {
    SEM_INIT(&cond[i], 0, 0) ;
  };
}
```

  - Ein die Ressource anfordernder Thread:

```
priority_t my_priority;
  ...
  allocate(my_priority); /* wait for resource */
    /* use resource */
  if(deallocate(my_priority) <= 0) {
      /* cannot deallocate resource, */
      /* undertake some recovery operation */
  }
```

Semaphore

- Kritik an Semaphoren
  - eher „low-level“ –Synchronisationsmethode
  - Fehleranfällig
    - der Programmierer ist verantwortlich für die technische Realisierung einer Lösung der Konsistenzproblem
    - Es kann leicht „etwas vergessen werden“ (Flüchtigkeitsfehler)
    - Versehentlicher ungeschützter Zugriff auf eine kritischen Bereich ist möglich
    - Es kann leicht zu deadlocks kommen
      - z.B. durch Vergessen eines Signal-Aufrufs
  - Wünschenswert: ein „high-level“-Ansatz zur Synchronisation
    - Gegenseitigen Ausschluss nicht programmieren, sondern deklarieren
    - Die korrekte Implementierung übernimmt das Laufzeitsystem

Monitore

- Idee [*Per Brinch-Hansen, Tony Hoare*]
  - Zu schützende Ressource (gemeinsame Daten) und Operationen zu deren Manipulation werden in einem Modul gekapselt: einem **Monitor**.
  - Die Implementierung der Daten/Ressourcen ist geheim
  - Die Operationen werden per Definitionem im gegenseitigen Ausschluss ausgeführt
    - Übersetzer bzw. Laufzeitsystem müssen den gegenseitigen Ausschluss (z.B. mit Hilfe von Semaphoren) realisieren

© Per Brinch-Hansen, 1999

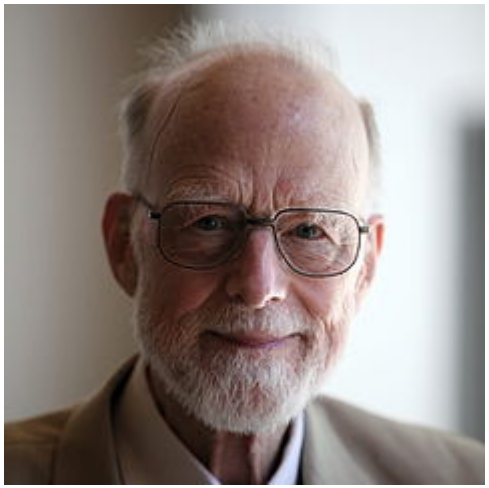
Per Brinch-Hansen
1939 - 2007

Pionier im Bereich Betriebssysteme und Nebenläufige Programmierung

Monitor-Konzept, Concurrent Pascal

2002: IEEE Computer Pioneer Award

Sir Tony Hoare, geb. 1934



© Rama, Wikimedia Commons, Cc-by-sa-2.0-fr

Britischer Informatik-Pionier mit großem Einfluss auf strukturierte Programmier-sprachen

Quicksort-Algorithmus
„Hoare-Kalkül“
CSP: Communicating Sequential Processes

Turing Award 1980

Monitore

- *Beispiel:* Struktur eines Monitors für einen begrenzten Puffer (Pseudocode)

```
monitor buffer;

  export append, take; -- Schnittstelle des Monitors

  -- Deklaration der erforderlichen Variablen
  ...
  -- Deklaration der Monitor-Operationen
  procedure append(I:integer); ... end;
  procedure take(out I:interger); ... end;

begin
  -- Initialisierung der Monitor-Variablen ...
end buffer;
```

Monitore

- Bedingungssynchronisation mit Monitoren [Hoare 1974]
  - ***Condition Variables*** (Bedingungsvariablen, Ereignisvariablen)
    - Sind innerer Bestandteil eines Monitors, ähnlich Semaphoren
    - Mit je einer eigenen Warteschlange
  - Operationen ***wait*** und ***signal***
    - **`wait(CV):`** *blockiere aufrufende Task;*
      aufrufende Task in CV-Warteschlage eintragen;
      falls Tasks auf den Zugriff auf den Monitor
      (→ gegenseitiger Ausschluss) warten,
      gib eine frei;
    - **`signal(CV):`** *gib den nächsten blockierten Task aus der CV-Warteschlagen frei, falls vorhanden;*
  - Beachte:
    - *wait(cv)* blockiert **immer** die aufrufende Task!
      - Semaphore: wait(S) blockiert nur, wenn S = 0
    - d.h. eine Task muss selbst prüfen, ob die Synchronisationsbedingung erfüllt ist, und nur falls nicht, *wait()* aufrufen (d.h. sich selbst blockieren)

Monitore

- *Beispiel:* Ein begrenzter Puffer als Monitor (Pseudocode)

```
monitor buffer;

  export append, take; -- Exportierte Operationen des Monitors

  -- Daten des Monitors (nicht nach außen sichtbar!)
  constant size = 32;
  buf: array[0...size-1] of integer;
  top, base: 0...size-1;
  SpaceAvailable, ItemAvailable: condition;
  NumberInBuffer: integer;

  procedure append(I: integer);
  begin
    if NumberInBuffer = size then wait(SpaceAvailable);
    buf[top] := I; NumberInBuffer := NumberInBuffer + 1;
    top := (top + 1) mod size;
    signal(ItemAvailable);
  end append;
```

- *Beispiel:* Ein begrenzter Puffer als Monitor (Forts.)

```
procedure take(out I: integer);
begin
  if NumberInBuffer = 0 then wait(ItemAvailable);
  I := buf[base]; NumberInBuffer := NumberInBuffer - 1;
  base := (base + 1) mod size;
  signal(SpaceAvailable);
end append;

begin -- Initialisierungen
  NumberInBuffer = 0;
  top := 0;
  base := 0;
end buffer;
```

Monitore

- Problem
  - Wenn nach einem **`signal()`** die aufrufende Task weiter rechnet, sind evtl. zwei Tasks im Monitor aktiv:
    - die `signal()` aufrufende und die dadurch frei gegebene
- Lösung: Erweiterung der Semantik von `signal()`
  - 4 alternative Möglichkeiten:
    - `signal()` ist nur erlaubt als letzte Aktion einer Task, bevor sie den Monitor verlässt, d.h. als letzte Anweisung in einer Monitor-Operation.
      - siehe Beispiel begrenzter Puffer
    - `signal()` beinhaltet als Seiteneffekt ein `return`, d.h. die aufrufende Task wird gezwungen den Monitor zu verlassen.
    - wenn `signal()` eine andere Task frei gibt, blockiert es die aufrufende Task. Diese wird fortgesetzt, sobald der Monitor frei ist [Hoare 1974].
      - die aufrufende Task wird in eine *ready queue* eingereiht
      - wenn der Monitor wieder frei ist, werden Tasks aus der *ready queue* den Tasks, die den Monitor neu betreten möchten, vorgezogen
    - die durch ein `signal()` frei zu gebende Task wird erst dann freigegeben, wenn die aufrufende Task den Monitor verlassen hat

Mutexe und Bedingungsvariablen in C/Real-Time POSIX

- **Monitore** können in C/Real-Time POSIX mit Hilfe von **Mutex-Variablen** und **Bedingungsvariablen** nachgebildet werden

  → Schnittstelle siehe Beispiel 3-2 und 3-3

- Ein Monitor ist assoziiert mit einer **Mutex-Variablen** (kurz: einem **Mutex**) mit **Mutex-Atttributen**

```
#typedef ... p_thread_mutex_t;
#typedef ... p_thread_mutexattr_t;
```

- **Initialisierung** von Mutex-Attributen und Mutexen

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Die Mutex-Atrribute **`*attr`** *werden mit Default-Werten initialisiert*

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

Der Mutex **`*mutex`** *wird mit den Mutex-Attributen* **`*attr`** *initialisiert*

Mutexe und Bedingungsvariablen in C/Real-Time POSIX

- Eine Monitor-Operation muss als erstes den **Mutex setzten** (***lock***) und als letztes den **Mutex** wieder **freigeben** (***unlock***)

```
int pthread_mutex_lock(pthread_mutex *mutex);
```

Falls der Mutex ***mutex** schon gesetzt ist, wird der aufrufende Thread blockiert.

Falls nicht, wird der Mutex gesetzt; der aufrufende Thread ist der Besitzer (*owner*) des Mutexes

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

Falls der aufrufende Thread der Besitzer (owner) des Mutex ***mutex** ist, wird der Mutex freigegeben und ein blockierter Thread wird freigegeben

– *Beispiel:*

- Grundsätzliches Vorgehen

```
// Deklaration der Variablen
pthread_mutex_attr_t attr;
pthread_mutex_t mutex;

// Initialisierung der Variablen
pthread_mutex_attr_init(&attr);
pthread_mutex_init(&mutex, &attr);
```

```
// Monitor-Operation
int monitor_operation(
                pthread_mutex_t *mutex ,...)
{ if(pthread_mutex_lock(mutex) != 0)
        return ERROR
  // Monitor-Operation ausführen ...
  if(pthread_mutex_unlock(mutex) != 0)
        return ERROR
  return 0;
}
```

Mutexe und Bedingungsvariablen in C/Real-Time POSIX

- Mutexe können ergänzt werden durch **Bedingungsvariablen** mit **Bedingungsvariablen-Attributen**

```
#typedef ... p_thread_cond_t;
#typedef ... p_thread_condattr_t;
```

- **Initialisierung** von Bedingungsvariablen und Bedingungsvariablen-Attributen

```
int pthread_condattr_init(pthread_condattr_t *cattr);
```

Die Bedingungsvariablen-Attribute **`*attr`** werden mit Default-Werten initialisiert

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *cattr);
```

Die Bedingungsvariable **`*cond`** wird mit den Bedingungsvariablen-Attributen **`*cattr`** initialisiert

Mutexe und Bedingungsvariablen in C/Real-Time POSIX

- Bedingungssynchronisation

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

Wenn der aufrufende Thread `*mutex` besitzt, wird er für `*cond` blockiert und `*mutex` wird freigegeben.

Im Erfolgsfall ist das Resultat 0; in Fehlerfällen ist das Resultat != 0;

Nach Rückkehr besitzt der aufrufende Thread wieder `*mutex`.

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

Gibt einen Thread frei, der für `*cond` blockiert ist.

– Empfohlene Verwendung innerhalb einer Monitor-Operation:

```
pthread_mutex_lock(&mutex);
...
while(condition_is_false)
  pthread_cond_wait(&cond,&mutex));
...
pthread_mutex_unlock(&mutex);
```

Nach Rückkehr aus `pthread_cond_wait()` sollte die Synchronisationsbedingung nochmals geprüft werden. Deshalb while-Schleife statt if-Anweisung

- *Beispiel:* Begrenzter Puffer als Monitor [Burns & Wellings 2009, Kap. 5.7]

```
#include <pthreads.h>

#define BUFF_SIZE 10

// Struktur für den Puffer
// beinhaltet auch den Mutex und
// die Bedingungsvariable
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t buffer_not_full;
  pthread_cond_t buffer_not_empty;
  int count, first, last;
  int buf[BUFF_SIZE];
  } buffer;
```

```
int initialize(buffer * B)
{
  /* Initialisierungen für
     den Mutex B->mutex und die
     Bedingungsvariablen
     B->buffer_not_full und
     B->buffer_not_empty */
  ...
}
```

- *Beispiel:* Begrenzter Puffer als Monitor (Forts.)

```
int append(int item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == BUFF_SIZE)/* Bedinung prüfen */
    PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex);
  /* put data in the buffer and update count and last */
  PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}

int take(int *item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->count == 0)
    PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
  /* get data from the buffer and update count and first */
  PTHREAD_COND_SIGNAL(&B->buffer_not_full);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}
```

Geschützte Objekte in Ada

- Ein **Geschütztes Objekt** (***protected object***) kapselt
  - Datenelemente,
  - **Unterprogramme**: Prozeduren, Funktionen.
  - **Eintrittspunkte** (***entries***): Prozeduren, deren Ausführung an eine explizite Vorbedingung (**Wächter**, ***guard***) geknüpft ist.
- Auf die Datenelemente eines geschützten Objekts darf nur durch die zugehörigen Unterprogramme oder Eintrittspunkte zugegriffen werden
- Das Laufzeitsystem muss garantieren, dass
  - die Unterprogramme und Eintrittspunkte so ausgeführt werden, dass die Manipulation der Datenelemente unter gegenseitigem Ausschluss erfolgt
  - Eintrittspunkte nur „betreten“ werden, wenn der Wächter *true* ist.

Geschützte Objekte in Ada

- Gegenseitiger Ausschluss mit geschützten Objekten
  - *Beispiel:* Geschützter Zugriff auf eine Integer-Variable

```
protected type Shared_Integer(Initial_Value:Integer) is
   function Read return Integer;
   procedure Write(New_Value: Integer);
   procedure Increment(By:Integer);
private
   The_Data: Integer := Initial_Value;
end Shared_Integegeger;

protected body Shared_Integer is
   function Read return Integer is
   begin return The_Data; end Read;

   procedure Write(New_Value:Integer) is
   begin The_Data := New_Value; end Write;

   procedure Increment(By:Integer) is
   begin The_Data := The_Data + By; end Increment;
end Shared_Integer;
```

Geschützte Objekte in Ada

- Bedingungssynchronisation mit Eintrittspunkten (*entries*)
  - *Beispiel:* Begrenzter Puffer

```
Buffer_Size: constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type  Bounded_Buffer is
   entry Get(Item: out Data_Item);
   entry Put(Item: in Data_Item);
private
   First: Index := Index'First;
   Last: Index := Index'Last;
   Number_In_Buffer : Count := 0;
   Buf: Buffer;
end Bounded_Buffer;
```

Geschützte Objekte in Ada

- Bedingungssynchronisation mit Eintrittspunkten (*entries*)
  - *Beispiel:* Begrenzter Puffer (Forts.)

```
protected body Bounded_Buffer is
  entry Get(Item: out Data_Item)
    when Number_In_Buffer /= 0 is
  begin
    Item := Buf(First); First := First+1;
    Number_In_Buffer := Number_In_Buffer - 1;
  end Get;
  entry Put(Item: in Data_Item) is
    when Number_In_Buffer /= Buffer_Size is
  begin
    Last := Last+1; Buf(Last) := Item;
    Number_In_Buffer := Number_In_Buffer + 1;
  end Put;
end Bounded_Buffer;
```

Geschützte Objekte in Ada

- Bedingungssynchronisation mit Eintrittspunkten (*entries*)
  - *Beispiel:* Ressourcensteuerung (vgl. 3-22ff)

```
protected type Ressource_Control is

   entry Allocate;
   procedure Deallocate;

private
   Free: Boolean := True;
end Resource_Control;

protected body Ressource_Control is

   entry Allocate when Free is
   begin Free := False; end Allocate;

   procedure Deallocate is
   begin Free := True; end Deallocate;

end Ressource_Control;
```

Literatur

[Burns & Wellings 2009] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, 2009.

[Brinch-Hansen 1973] Per Brinch-Hansen: *Operating Systems Principles.* Englewood Cliffs: Prentice Hall, 1973.

[Hoare 1974] Tony Hoare: *Monitors: An Operating Systems Structuring Concept*. Communications of the ACM, 1978.

[Wörn & Brinkschulte 2005] Heinz Wörn, Uwe Brinkschulte: *Echtzeitsysteme*. Springer, 2005.

[Zöbel 2008] Dieter Zöbel: *Echtzeitsysteme. Grundlagen der Planung*. Springer, 2008.