Synchronisation mit Path Expressions

Thao-Nguyen Do

Konzepte von Betriebssystemkomponenten
Friedrich-Alexander-Universität Erlangen-Nürnberg

04.Juli 2013

Nachteile der konventionellen Notation

```
Functions calling this function: down

  File                    Function                    Line
0 mtip32xx.c              mtip_hw_get_scatterlist     2687
1 cdma.c                  host1x_cdma_wait_locked      165
2 mthca_cmd.c             mthca_cmd_poll               350
3 mthca_cmd.c             mthca_cmd_wait               420
4 mthca_cmd.c             mthca_cmd_use_events         585
5 mthca_cmd.c             mthca_cmd_use_polling        600
6 hp_sdc.c                hp_sdc_init                  909
7 hp_sdc.c                hp_sdc_register             1058
8 hp_sdc.c                hp_sdc_register             1073

* 354 more lines - press the space bar to display more *
```

Nachteile der konventionellen Notation

```
Functions calling this function: up

  File                    Function                    Line
0 salinfo.c               salinfo_work_to_do           197
1 osl.c                   acpi_os_signal_semaphore    1284
2 mtip32xx.c              mtip_command_cleanup         183
3 mtip32xx.c              mtip_timeout_function        644
4 mtip32xx.c              mtip_async_complete          732
5 mtip32xx.c              mtip_async_complete          734
6 mtip32xx.c              mtip_hw_release_scatterlis  2662
7 mtip32xx.c              mtip_hw_get_scatterlist     2691
8 mtip32xx.c              mtip_hw_get_scatterlist     2695

* 714 more lines - press the space bar to display more *
```

Nachteile der konventionellen Notation

- Synchronisation über das ganze Programm verteilt
  - schwierig zu überblicken
  - nicht leicht zu verstehen
- Werkzeuge sehr primitiv (P/V, Signal/Wait)

Gliederung

1. Path Expression
   - Grundlagen
   - Stack
   - Vergleich mit Semaphoren
2. Implementierung
   - Allgemein
   - Algorithmus
   - Controller
3. Zusammenfassung
   - Weiterführendes

Path Expression

- zusammenhängende und einheitliche Notation von Synchronisation
- beschreiben Synchronisationsverhalten
- Spezifikation in Typendefinitionen
- Implementierung durch Controller
- basieren auf regulären Ausdrücken
- Beispiel: `path push(), pop(), {peak()} end`

Typendefinition

- Beschreibung der inneren Struktur
  - öffentliche Schnittstellen
  - zusätzlich: Synchronisationsverhalten der Prozeduren
- Datenobjekt abgeleitet von Typendefinitionen
- Trennung von Spezifikation und Implementierung
  - erleichtere Verifizierbarkeit
  - Wartungsfreundlichkeit
  - einfache Anpassbarkeit

Beispiel: Stack

```
class Stack{

   void push(int value); //Puts an element on top
   int pop();            //Removes and returns element on top
   int peek();           //Returns element on top

}
```

Gewünschtes Verhalten

- `push()`, `pop()`, `peek()` nicht gleichzeitig
- `peek()` mehrmals gleichzeitig
- Wiederverwendung der Path Expressions

Beispiel: Stack

```
class Stack{

   void push(int value); //Puts an element on top
   int pop();            //Removes and returns element on top
   int peek();           //Returns element on top

}
```

Gewünschtes Verhalten

- `push()`, `pop()`, `peek()` nicht gleichzeitig
- `peek()` mehrmals gleichzeitig
- Wiederverwendung der Path Expressions

Gegenseitiger Ausschluss

`push()`, `pop()`, `peek()` nicht gleichzeitig

Gegenseitiger Ausschluss

`push(), pop(), peek()`

Gegenseitiger Ausschluss

- Trennung durch ", "(Komma)
- keine Bevorzugung
- `push, pop, peek` nie gleichzeitig möglich

Gleichzeitige Ausführung

`peek()` mehrmals gleichzeitig

Gleichzeitige Ausführung

`push(), pop(), {peek()}`

Gleichzeitige Ausführung

- Notation "{" und "}"
- beschränkt auf benötige Aufrufe
- Parallele Ausführung von peek

Wiederholung

Wiederverwendung der Path Expressions

Wiederholung

```
path push(), pop(), {peek()} end
```

Wiederholung

- Notation "**path**" und "**end**"
- Path Expression kann immer wieder durchlaufen werden
- ermöglicht mehrmalige Ausführung der Path Expression

Stack mit Path Expressions

```
class Stack{

    Path Expressions{
      path push(), pop(), {peek()} end;
    }

   void push(int e); //Puts an element on top
   int pop();        //Removes and returns element on top
   int peek();       //Returns element on top

}
```

Semaphore mit Path Expressions

```
class Semaphore{

    Path Expressions{
        path {V() , P()} end;
    }

    int counter;

    void V();
    void P();
    void init(int value);
}
```

- ausgeführte P() nie größer als ausgeführte V()
- Zählervariable nur durch P() und V() modifizierbar
- Semaphor instanziierbar

⇒ gleiche Mächtigkeit wie primitive P()-/V()-Operationen

Semaphore mit Path Expressions

```
class Semaphore{

    Path Expressions{
        path {V() , P()} end;
    }

    int counter;

    void V();
    void P();
    void init(int value);
}
```

- ausgeführte P() nie größer als ausgeführte V()
- Zählervariable nur durch P() und V() modifizierbar
- Semaphor instanziierbar

$\Rightarrow$ gleiche Mächtigkeit wie primitive P()-/V()-Operationen

Konzept

Allgemein

- Erstellung der Controller durch Compiler möglich
- Verwendung eines rekursiven Algorithmus
- Nutzung von primitiveren Synchronisationsmethoden
- Erweiterung der Funktionen um Prolog und Epilog

Vorgehen

- Erstellung von Prolog und Epilog für Funktionen
- Schlösser (Locks) initialisieren
- Prolog mit Schlossanfrage
- Epilog mit Schlossfreigabe

Beispiel

```
path push(), pop(), {peek()} end
```

Wiederholung: **path** und **end**

- Semaphore `S1` erstellen
- `path` ersetzen mit `P(S1)`
- `end` ersetzen mit `V(S1)`

Beispiel

```
P(S1) push(), pop(), {peek()} V(S1)
```

Beispiel

```
P(S1) push(), pop(), {peek()} V(S1)
```

Gegenseitiger Ausschluss: ","

- Jede Funktion mit der äußeren Semaphore umgeben

Beispiel

```
P(S1) push() V(S1)        P(S1) pop() V(S1)
                P(S1) {peek()} V(S1)
```

Beispiel

```
P(S1) push() V(S1)        P(S1) pop() V(S1)
                P(S1) {peek()} V(S1)
```

Funktionsnamen

- `push()` und `pop()` bestehen nur noch aus dem Funktionsnamen
- alle Operationen vor Funktionsnamen in den Prolog
- alle Operationen nach Funktionsnamen in den Epilog

Beispiel

```
P(S1) push() V(S1)      P(S1) pop() V(S1)
              P(S1) {peek()} V(S1)
```

Gleichzeitige Ausführung

- Zusätzliche Semaphore S2 und Zählervariable C1 initialisieren
- P(S1) ersetzen mit PP(C1, S2, S1)
  Ausschluss der anderen Funktionen durch Blockierung von S1 und Ausschluss beim Hochzählen des Counters C1 durch S2
- V(S1) ersetzen mit VV(C1, S2, S1)
  analoge Funktionsweise zu PP()

Beispiel

```
P(S1) push() V(S1)        P(S1) pop() V(S1)
     PP(C1, S2, S1) peek() VV(C1, S2, S1)
```

Beispiel: Ergebnis

```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

```
int peek(){
  PP(C1, S2, S1);  //Prolog
  // Function body
  return value;
  VV(C1, S2, S1);  //Epilog
}
```

- P()- und V()-Operationen umschließen gesamte Funktion
- Funktionsausführung geht nach return in den Epilog über

Funktionsweise des Controllers

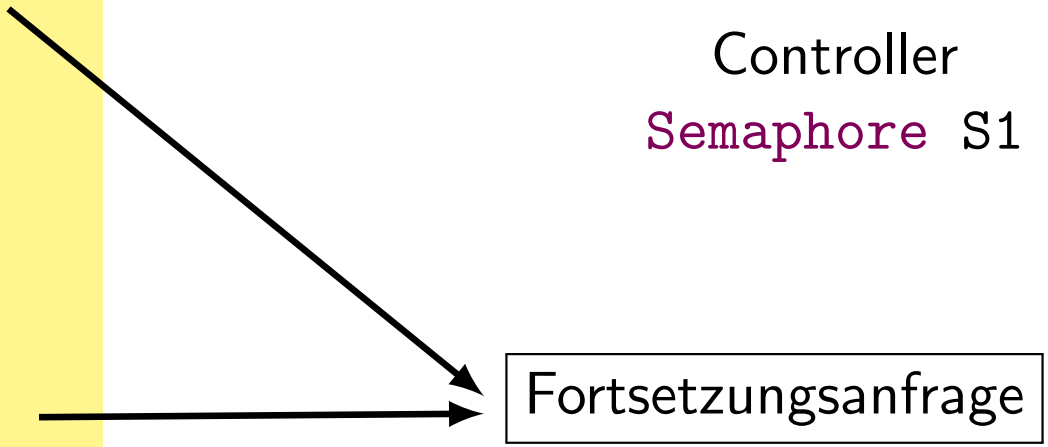
```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

```
path push, pop end
```

Controller

```
Semaphore S1
```



Funktionsweise des Controllers

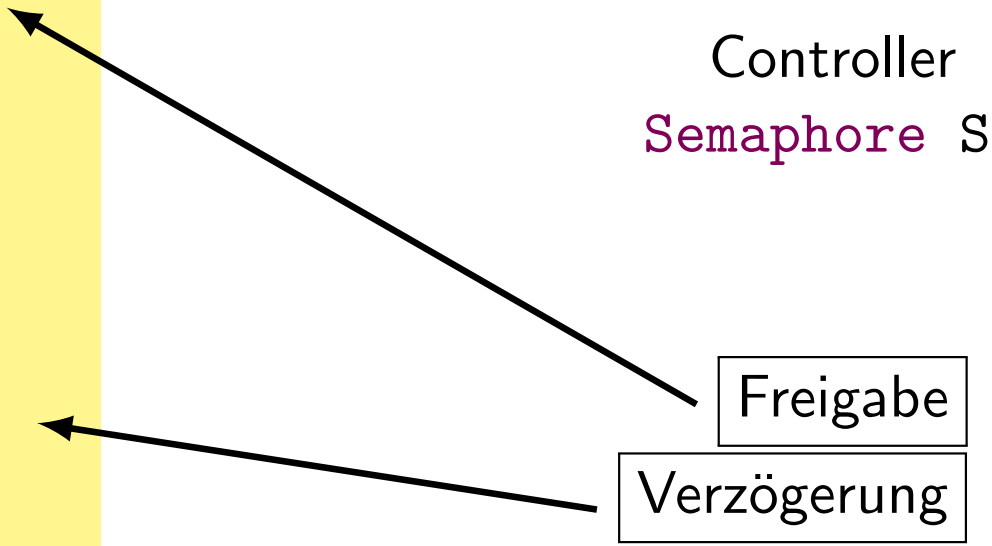
```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

```
path push, pop end
```

Controller

```
Semaphore S1
```



Funktionsweise des Controllers

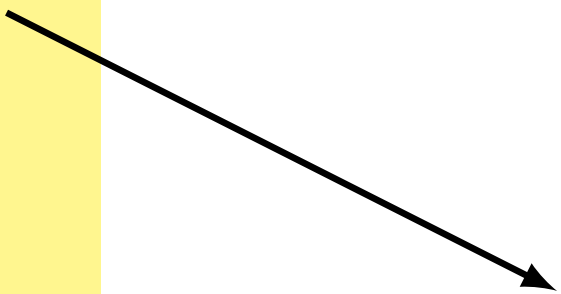
```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

`path push, pop end`

Controller

`Semaphore S1`



Benachrichtigung: Ausführung `push` vollendet

Funktionsweise des Controllers

```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

```
path push, pop end
```

Controller

```
Semaphore S1
```

Funktionsweise des Controllers

```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

`path push, pop end`

Controller

`Semaphore S1`

Benachrichtigung: Ausführung pop vollendet

Funktionsweise des Controllers

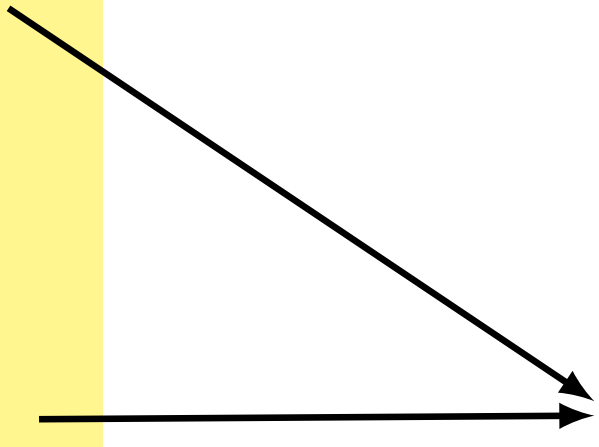
```
void push(int value){
  P(S1);     // Prolog
  // Function body
  return;
  V(S1);     // Epilog
}

int pop(){
  P(S1);     // Prolog
  // Function body
  return value;
  V(S1);     // Epilog
}
```

Path expression

```
path push, pop end
```

Controller

```
Semaphore S1
```



Zusammenfassung

- zusammenhängende und einheitliche Synchronisationsnotation
- Path Expressions repräsentieren erlaubtes Synchronisationsverhalten
- tatsächliche Synchronisationsimplementierung austauschbar

- gleiche Mächtigkeit wie P-/V-Operationen
- Verifikation durch endliche Zustandsautomaten
- automatische Codeerstellung möglich

$\Rightarrow$ Code ist weniger fehleranfällig

Zusammenfassung

- zusammenhängende und einheitliche Synchronisationsnotation
- Path Expressions repräsentieren erlaubtes Synchronisationsverhalten
- tatsächliche Synchronisationsimplementierung austauschbar

- gleiche Mächtigkeit wie P-/V-Operationen
- Verifikation durch endliche Zustandsautomaten
- automatische Codeerstellung möglich

$\Rightarrow$ Code ist weniger fehleranfällig

Zusammenfassung

- zusammenhängende und einheitliche Synchronisationsnotation
- Path Expressions repräsentieren erlaubtes Synchronisationsverhalten
- tatsächliche Synchronisationsimplementierung austauschbar

- gleiche Mächtigkeit wie P-/V-Operationen
- Verifikation durch endliche Zustandsautomaten
- automatische Codeerstellung möglich

$\Rightarrow$ Code ist weniger fehleranfällig

Weiterführendes

Warum werden Path Expressions nicht häufiger verwendet?

- nicht effizient, gesamte Funktion blockiert?
- komplexe Synchronisation schwierig darzustellen
  z.B. viele Funktionen

Wie können Path Expressions verbessert werden?

- lokaler Einsatz (global zu komplex)
- Synchronisation aus mehreren einzelnen Path Expressions aufbauen
  $\rightarrow$ Verlust der Übersichtlichkeit

Mindestkompromiss: gute Dokumentation des Quellcodes