2008

Vorige Keer

- **mutual exclusion**
- **kritieke sectie (cs)**

mbv

1. software : Dekker's, Peterson's algoritme
2. hardware: *uniprocessor machine*: disable interrupts
3. hardware: *multiprocessor machine*: speciale machineinstructies, bijv. set&test

- Nadeel van 1 & 3 : **busy waiting**
- Nadeel van 2 : **bloedlink**
- Daarom in Hfst 5.3: **semaforen [Dijkstra'65]**

Binaire Semaforen

- Binaire semafoor **s** bestaat uit:
  - *integer* **s.count**, met waarde 0 of 1
  - een *rij van geblokkeerde processen* **s.queue**
  - 2 *atomaire* operaties: **wait** en **signal**
- Idee:

  count $= 0 \iff$ kritieke sectie bezet

  proces wacht in **queue** tot het naar binnen mag
- **wait(s)**
  - als je mag ga cs in en zet **s.count** op 0
  - anders blokkeer in **s.queue**
- **signal(s)**
  - wek indien mogelijk proces in **s.queue**
  - anders zet **s.count** terug op 1

Mutual Exclusion

```
P1              P2
⋮               ⋮
wait(s);        wait(s);
cs1;            cs2;
signal(s);      signal(s);
⋮               ⋮
```

- binaire semafoor wordt ook wel **lock** genoemd
- wait is dan **acquire lock**
- signal is dan **release lock**

Implementatie

```
struct semaphore {
        int count;
        list_of_processes queue;
        }

void wait(semaphore s)
{ if (s.count = 1 )
     then { s.count = 0 }
     else { zet dit proces in s.queue;
            blokkeer het }
}

void signal(semaphore s)
{ if empty(s.queue)
     then { s.count = 1 }
     else { haal een proces uit s.queue ;
            zet het in ready queue }
}
```

Implementatie

- Maar: wait en signal moeten **atomair** zijn!
- Dus: **mutual exclusion** nodig!
- Oplossingen:
  - software oplossing (Dekker's, Peterson's)
    probleem: overhead, beetje busy waiting
  - op uniprocessor machine: disable interrupts
    redelijk, gegeven korte en vaste duur operaties
  - gebruik speciale machineinstructie (set&test)
    probleem: beetje busy waiting
  - wait en signal zelf als machine-instructies?

Voordelen van Semaforen

- Busy waiting ?
  - Neen: proces wordt gewekt en hoeft niet zelf te kijken of het weer verder mag
- Starvation ?
  - Als het wekken via First-In-First-Out gaat zal er geen starvation optreden
  - Bij een ander “wekbeleid” mogelijk wel starvation
- Ten opzichte van uitschakelen interrupts:
  - Semaforen niet zo link
  - Verschillende semaforen worden voor verschillende soorten kritieke secties gebruikt

Algemene Semaforen

- Algemene semafoor **s** bestaat uit
  - *integer* **s.count**, met initiële waarde ≥ 0
  - een *rij van geblokkeerde processen* **s.queue**
  - *atomaire* operaties: **wait**, **signal**
- Idee: **count** is het aantal processen dat nog naar binnen mag
- **wait(s)**
  - verlaag eerst **s.count**
  - blokkeer in **s.queue** als **s.count** < 0
- **signal(s)**
  - verhoog eerst **s.count**
  - wek proces in **s.queue** als **s.count** ≤ 0

Implementatie

```
struct semaphore {
        int count;
        list_of_processes queue;
        };

void wait(semaphore s)
{ s.count = s.count-1;
  if (s.count<0) {zet dit proces in s.queue;
                  blokkeer het}
};

void signal(semaphore s)}
{ s.count = s.count+1;
  if (s.count<=0) {haal ander proces uit s.queue;
                   zet het in ready queue}
};
```

- Natuurlijk weer atomair!

Eigenschappen

- Het wekken van een proces wil zeggen dat het in de READY-queue wordt geplaatst; het hoeft dus niet meteen aan de beurt te zijn om instructies uit te voeren
- Semaforen worden niet uitsluitend gebruikt om kritieke secties af te schermen, maar meer algemeen om ervoor te zorgen dat bepaalde code alleen wordt uitgevoerd als aan bepaalde eisen is voldaan.

Producer/Consumer Probleem

- *n* producers en 1 consumer van buffer `b`
- Gemeenschappelijke variabelen: `in, out, b`
- **producer**

```
while (true) { "produceer item v";
                b[in] = v;
                in = in+1;
             }
```

- **consumer**

```
while (true) { "wacht tot in > out";
               w = b[out];
               out = out+1;
               "verwerk item w";
             }
```

- Wat zijn de synchronisatie-problemen hier ?

Oneindige Buffer

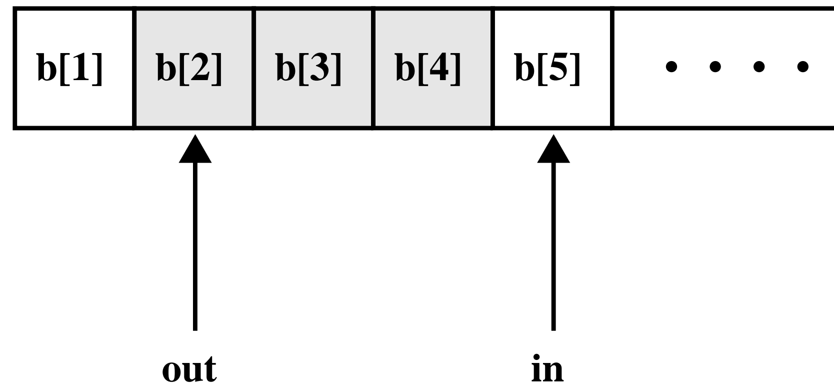


Figure 5.12 Infinite Buffer for the Producer/Consumer Problem

Oplossing?

- bin. sem. `s`, initieel `s.count=1`
- bin. sem. `delay`, initieel `delay.count=0`
- **producer**

```
while (true) { produce();
               semWaitB(s);
               append();
               n++;
               if (n==1) {semSignalB(delay)};
               semSignalB(s);
             }
```

- **consumer**

```
semWaitB(delay);
while (true) { semWaitB(s);
               take();
               n--;
               semSignalB(s);
               consume();
               if (n==0) {semWaitB(delay)};
             }
```

Zie Uppaal tegenvoorbeeld.

Consumer Aanpassing

- Introduceer hulpvariabele

```
semWaitB (delay);
while (true) { semWaitB(s);
               take();
               n--;
               m = n;
               semSignalB(s);
               consume();
               if (m==0) {semWaitB(delay)}
             }
```

- Nu wel beide problemen opgelost!

Oplossing met Alg. Semaforen

- bin. sem. `s`, initieel `s.count=1`
- alg. sem. `n`, initieel `n.count=0`
- **producer**

```
while (true) { produce();
               semWait(s);
               append();
               semSignal(s);
               semSignal(n);
             }
```

- **consumer**

```
while (true) { semWait(n);
               semWait(s);
               take();
               semSignal(s);
               consume();
             }
```

Producer Aanpassing?

- Verwissel `signal(s)` en `signal(n)`
- **producer**

```
while (true) { produce();
               semWait(s);
               append();
               semSignal(n);
               semSignal(s);
             }
```

- **consumer**

```
while (true) { semWait(n);
               semWait(s);
               take();
               semSignal(s);
               consume();
             }
```

Producer Aanpassing?

- Verwissel `signal(s)` en `signal(n)`
- OK omdat consumer op beide semaforen wacht

Consumer Aanpassing?

- Verwissel `wait(n)` en `wait(s)`
- **producer**

```
while (true) { produce();
               semWait(s);
               append();
               semSignal(n);
               semSignal(s);
             }
```

- **consumer**

```
while (true) { semWait(s);
               semWait(n);
               take();
               semSignal(s);
               consume();
             }
```

Consumer Aanpassing?

- Verwissel `wait(n)` en `wait(s)`
- **producer**

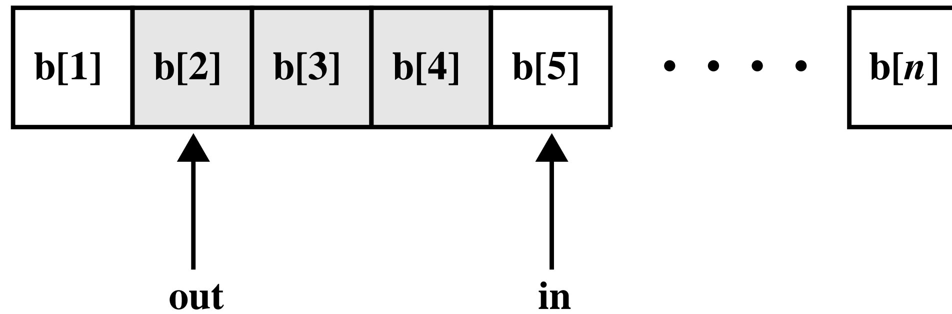
```
while (true) { produce();
               semWait(s);
               append();
               semSignal(n);
               semSignal(s);
             }
```

- **consumer**

```
while (true) { semWait(s);
               semWait(n);
               take();
               semSignal(s);
               consume();
             }
```

- Deadlock als consumer in cs bij lege buffer

Eindige Buffer



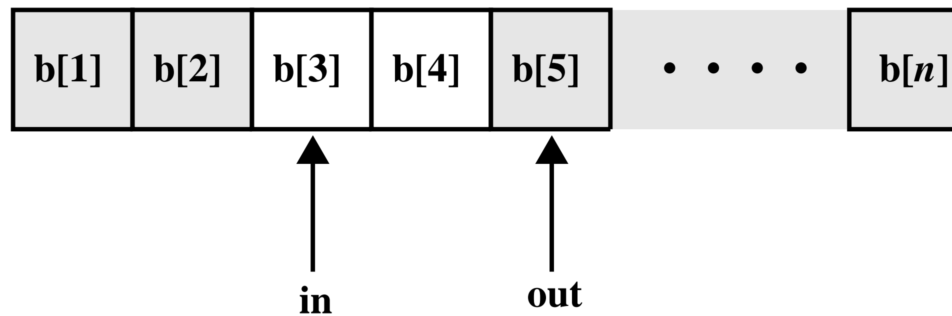


Figure 5.16 Finite Circular Buffer for the Producer/Consumer Problem

Oplossing?

- bin. sem. `s`, initieel `s.count=1`
- alg. sem. `n`, initieel `n.count=0`
- alg. sem. `e`, initieel `e.count=BUFFERSIZE`
- **producer**

```
while (true) { produce();
               wait(e);
               wait(s);
               append();
               signal(s);
               signal(n); }
```

- **consumer**

```
while (true) { wait(n);
               wait(s);
               take();
               signal(s);
               signal(e); }
```

Oplossing?!!!

- bin. sem. `s`, initieel `s.count=1`
- alg. sem. `n`, initieel `n.count=0`
- alg. sem. `e`, initieel `e.count=BUFFERSIZE`
- **producer**

```
while (true) { produce();
               wait(e);
               wait(s);
               produce();
               signal(s);
               signal(n); }
```

- **consumer**

```
while (true) { wait(n);
               wait(s);
               take();
               signal(s);
               signal(e); }
```