ICT Infrastructuren: Concurrency en Deadlock

25 november 2013

David N. Jansen

Opdracht 1, vraag

- Wie is het gelukt te assembleren/compileren?
- Wie heeft al een paar instructies geschreven?

- Mac OS X: bug + workaround.

Leerdoelen voor vandaag

- concurrency, interleaving
- mutual exclusion
  als oplossing voor interferentie
- deadlock (bijwerking van mutual exclusion)
  en wat je ertegen kunt doen

- over twee weken: toepassing in een concurrency lab (FSP)

Concurrency

- Meerdere processen worden (bijna) te gelijker tijd uitgevoerd

- voorbeelden
  - meerdere programma's
  - één programma met threads
  - binnen het besturingssysteem
    vaak per taak één proces

Interleaving

- acties van meerdere processen door elkaar
- niet altijd precies afwisselend
  - processen zijn soms niet even snel

- voorbeeld in FSP-notatie:
  P = (a → b → stop) Q = (c → d → stop)
  - a → b → c → d → stop
  - a → c → d → b → stop
  - c → a → d → b → stop
  - a → c → b → d → stop
  - c → a → b → d → stop
  - c → d → a → b → stop

Samenwerking

- sommige processen denken dat ze alleen zijn
  - gebruik van niet-verdeelbare bronnen?
- andere communiceren met elkaar
  - impliciet: gedeelde variabelen/bestanden
  - expliciet: berichten sturen

Kritieke sectie en interferentie

- sommige proces-delen zijn „kritiek”: meerdere acties (→ onderbreking mogelijk), terwijl exclusieve toegang tot bron vereist is
  - tijdelijk geen toegang voor andere processen
  - anders: interferentie = ongewenst gedrag als gevolg van (een bepaalde) interleaving

- voorbeeld
  - i := 2 en daarna parallel (i := i + 1) || (i := 2 * i)

Mutual Exclusion

- de standaardoplossing voor interferentie

- idee: een bepaalde bron mag niet door iedereen tegelijk gebruikt worden
- iemand moet ervoor zorgen dat beperkingen in acht genomen worden
  - wie?

Taken van besturingssysteem

- bronnen verdelen en beschermen
  (processor, geheugen, in-/uitvoer, bestanden)
- processen bewaken

- doel: geen interferentie
- zelfs een solitair proces
  moet soms rekening houden met anderen!

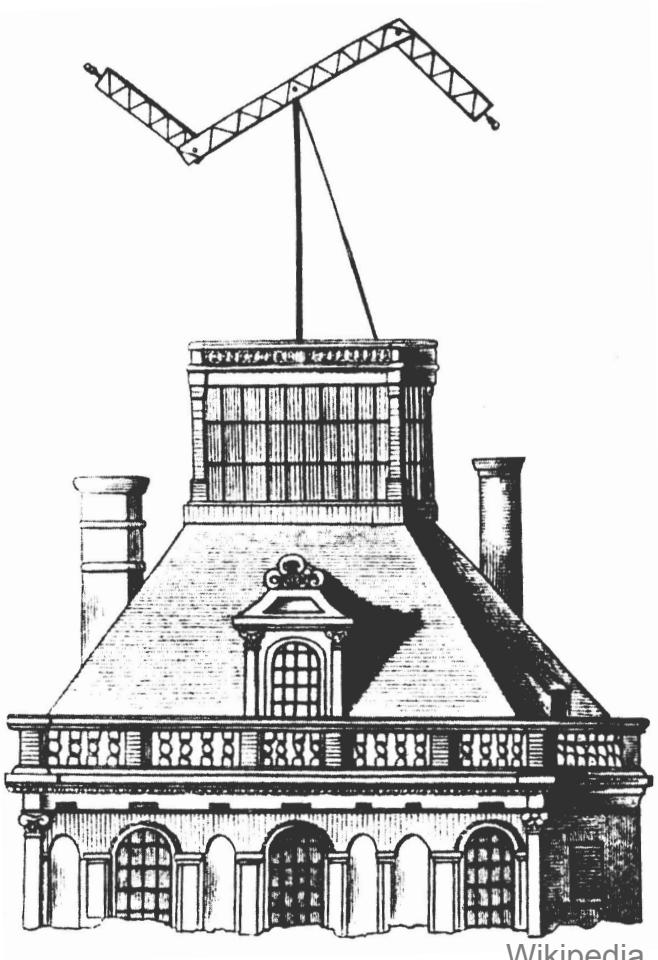
Mutex in de praktijk

- Drie manieren om mutex te implementeren:
  - Processor / hardware
  - Semaforen
  - Monitoren

Oefening

- In groepjes deze drie manieren behandelen
- elke groep kiest één manier
  en legt haar aan de anderen uit
  - special machine instructions, pp. 230–232
  - semaphores, pp. 233–234; figuur 5.7
  - monitors, pp. 246–248; figuur 5.15

Een oude semafoor: de Chappe-telegraaf



Wikipedia

Mutex: semaforen

- semafoor = seinpaal
- speciale variabele, altijd ≥ 0

- P(var) probeer de variabele te verlagen (en wacht zo nodig tot dat kan)
- V(var) verhoog de variabele

Dijkstra: http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF

Stallings: P = semWait, V = semSignal,

variabele kan wel < 0 worden, maar processen moeten dan wachten

Voorbeeld: Semafoor

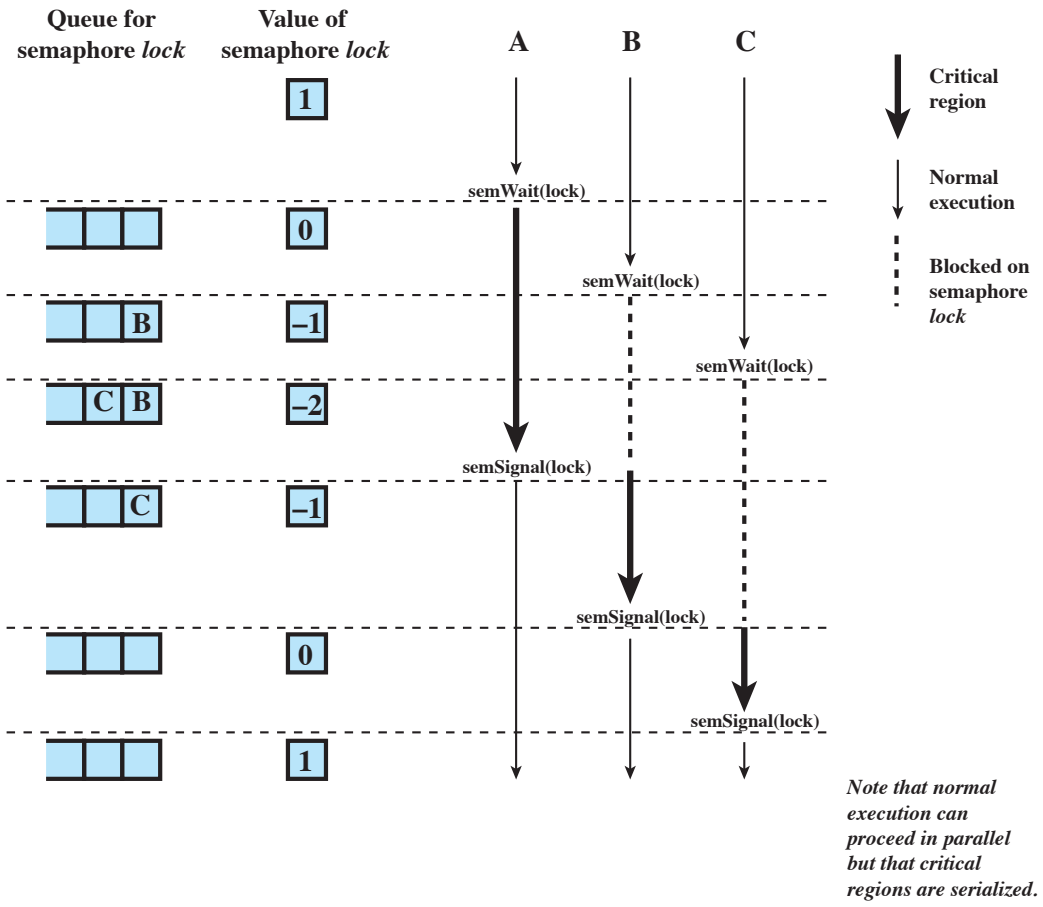


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Mutex: monitoren

- speciale subroutine met exclusieve rechten
- wie kritieke bronnen wil gebruiken
  moet dat via een monitor doen
- monitor zorgt voor mutex

- voordeel over semafoor:
  - gebruiker van kritieke bron vergeet nooit V(var)

Monitor

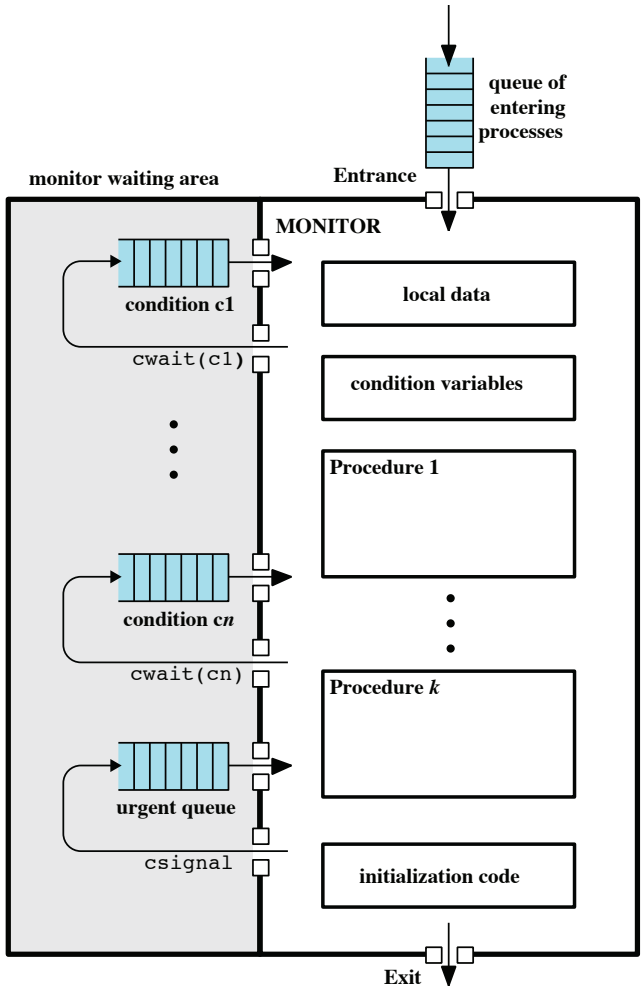


Figure 5.15 Structure of a Monitor

Mutex: processor / hardware

- interrupts tijdelijk verbieden
  - werkt alleen voor korte kritieke secties
  - werkt niet bij multicore/multiprocessor-systemen
- speciale instructies, b.v. in x86
    - lock-instructie: maak volgende instructie atomair
    - xchg-instructie: verwissel geheugenwoord met register
  - deze instructies kunnen niet onderbroken worden, niet eens in multiprocessor-systemen
  - eigenlijk te primitief, nog geen volwaardig mutex

Hardware support for mutual exclusion

Compare and swap

```
int compare_and_swap(
        int *word,
        int testval,
        int newval)
{
        int oldval;
        oldval = *word;
        if ( oldval == testval )
        *word = newval;
        return oldval;
}
```

Exchange

```
void exchange(
        int *register,
        int *memory)
{
        int temp;
        temp = *memory;
        *memory = *register;
        *register = temp;
}
```

Hardware support for mutual exclusion

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
     while (compare_and_swap(&bolt, 0, 1) == 1)
         /* do nothing */;
     /* critical section */;
     bolt = 0;
     /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```

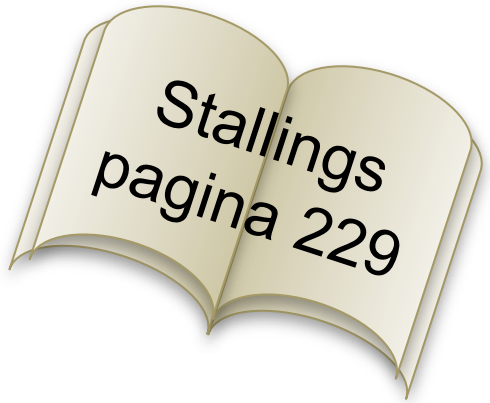
(a) Compare and swap instruction

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (&keyi, &bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Eisen aan mutex



- Besturingssysteem moet zorgen voor:
  - ① slechts één proces tegelijk in kritieke sectie
  - ② als een proces in de niet-kritieke sectie stopt, heeft dat geen gevolg voor de andere
  - ③ een proces dat toegang vraagt krijgt die
  - ④ als geen proces toegang heeft, krijgt de aanvrager onmiddellijk toegang
  - ⑤ geen aannames over snelheid van processen
- Processen moeten zorgen voor:
  - ⑥ niet oneindig lang in kritieke sectie blijven

Bijwerkingen

- Deadlock: dodelijke omarming
- Livelock: systeem rommelt,
  maar doet niets productiefs
- Starvation: één proces wacht oneindig lang

Deadlock in detail

- statische voorwaarden voor deadlock
  - mutual exclusion
  - bronnen vasthouden en wachten
  - geen preëmptie
    (d.w.z. OS kan niet dwingen tot tijdelijk afstaan)
- dynamische voorwaarde voor deadlock
  - cirkel van wachtende processen

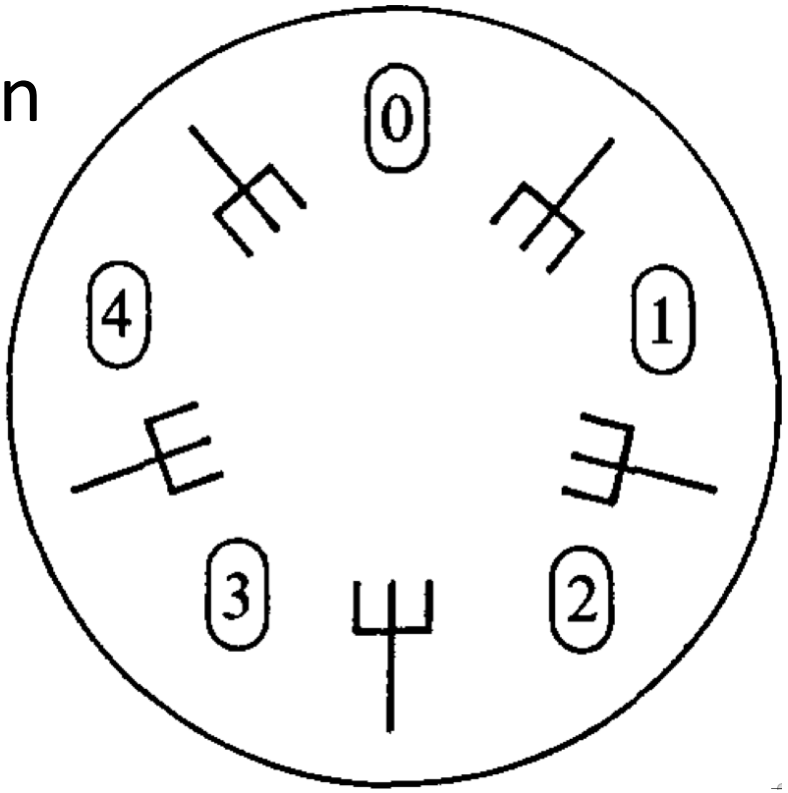
- alle vier voorwaarden vervuld → deadlock!

Vijf filosofen

- klassiek probleem om deadlock te illustreren

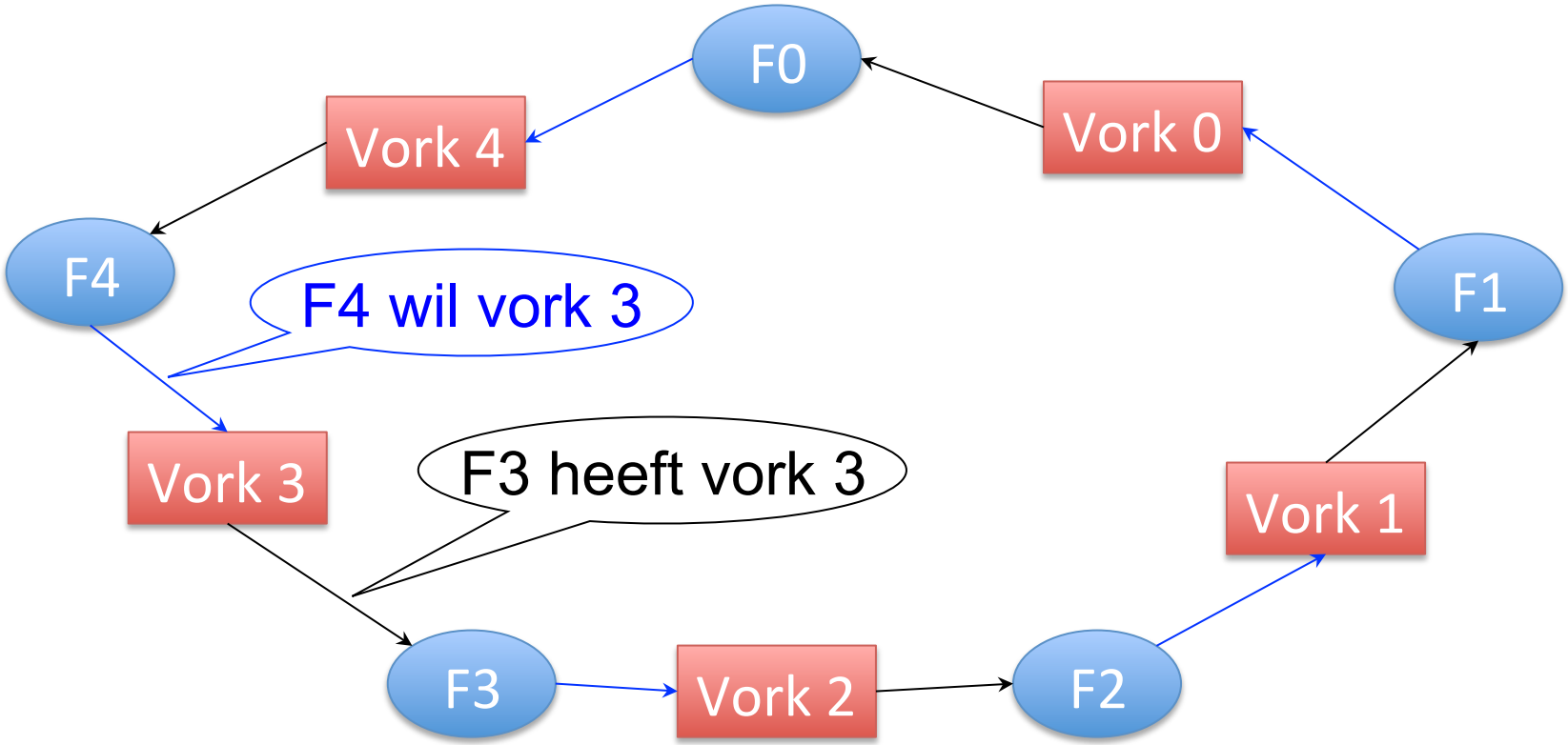
- 5 filosofen denken en eten
- eten met twee vorken
- 5 vorken

- [E. W. Dijkstra: Hierarchical ordering of sequential processes. Acta Informatica 1, 115–138. 1971]



Cirkel van wachtende processen

- graaf: pijlen tussen processen en bronnen



Maatregelen tegen Deadlock

- Preventie: verander één van de voorwaarden (statisch)
- Avoidance:
  beïnvloed dynamiek/snelheid van processen
- Detectie: laat de zaak op haar beloop gaan
  en grijp achteraf in

- praktijk: combinatie, afhankelijk van bron

Voorkómen (= Preventie) van Deadlocks

- één van de vier voorwaarden veranderen
  - geen mutual exclusion aanbieden
  - geen bronnen sparen, maar alles tegelijk aanvragen
  - preëmptie toestaan
  - ... of ...
  - bronnen in een vaste volgorde aanvragen (voor elk proces dezelfde)

Ontwijken (=Avoidance) van Deadlocks

- processen moeten van te voren aangeven hoeveel bronnen ze maximaal nodig hebben
- het OS laat een proces alleen toe als er voldoende bronnen over zijn
  - testen bij processtart
  - of: testen bij elke aanvraag van extra bronnen
  - Bankiers-algoritme: kan ik dit proces nog krediet verlenen zonder kans te lopen failliet te gaan? (nalezen in Stallings!)

Detecteren van deadlocks

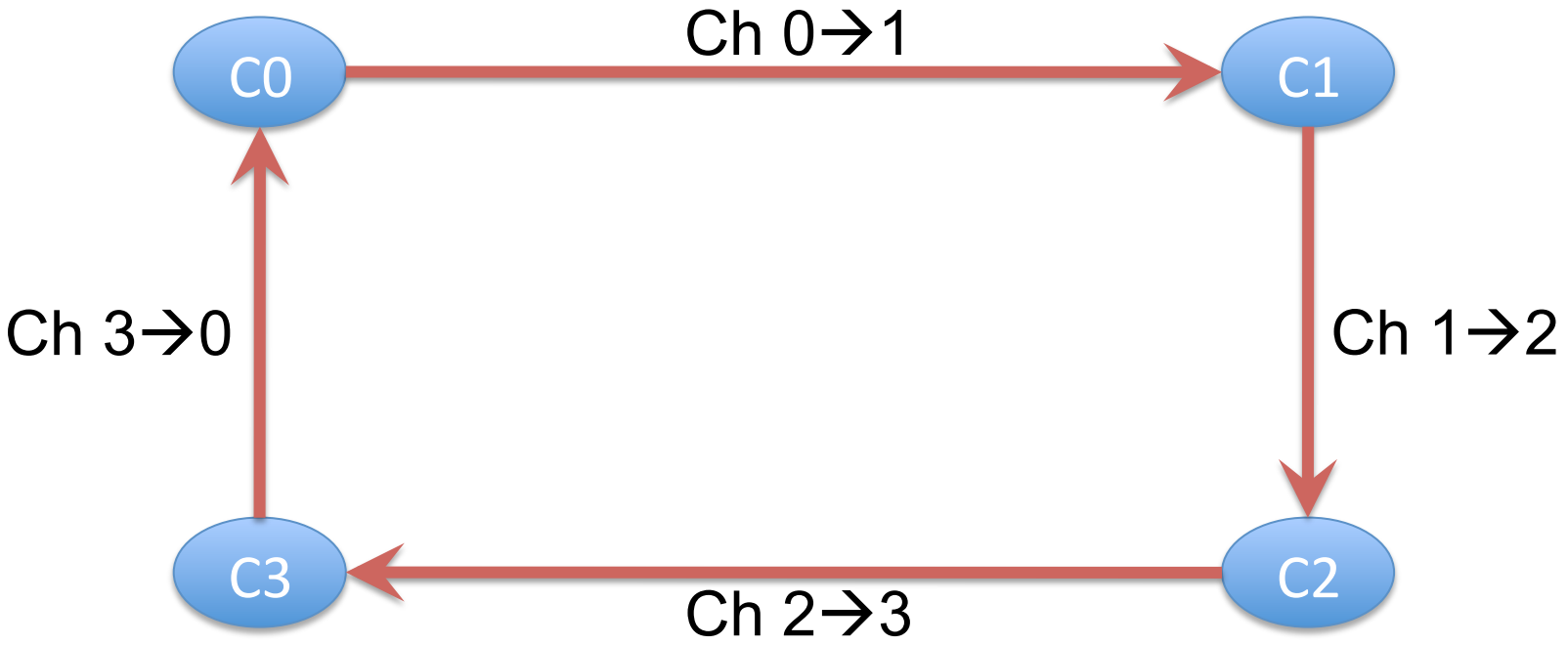
- OS doet niets om deadlocks te vermijden
- OS test regelmatig of deadlock is ontstaan
- zo ja: processen afbreken

Samenvatting

- concurrency
- concurrency in kritieke procesdelen leidt tot interferentie
- remedie: mutual exclusion
- bijwerking: deadlock, starvation
- maatregelen tegen deadlock

Network-on-chip

Beschouw een chip met vier cores. Deze vier cores moeten elkaar berichten kunnen zenden. Daartoe wordt een aantal communicatiekanalen tussen de cores aangelegd. In eerste instantie als volgt:

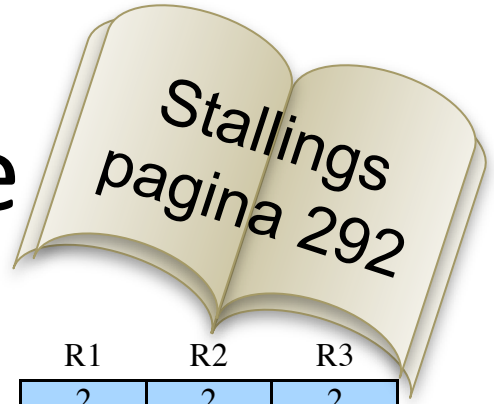


Network-on-chip

Elk communicatiekanaal is tegelijk een buffer voor één bericht. Als het bericht dus niet voor de directe buur-core bedoeld is moet eerst het volgende communicatiekanaal leeggemaakt worden voordat het bericht doorgestuurd wordt.

In deze configuratie van communicatiekanalen zijn deadlocks mogelijk. Laat zien hoe. Maak ook een voorstel om deadlocks te verhinderen zonder dat berichten verlorengaan.

Deadlock avoidance



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Deadlock avoidance

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Deadlock avoidance

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Deadlock avoidance

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Deadlock avoidance: een onveilige toestand

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State