Parallella Processer 3/9

Klassiska problem

Partyproblemet

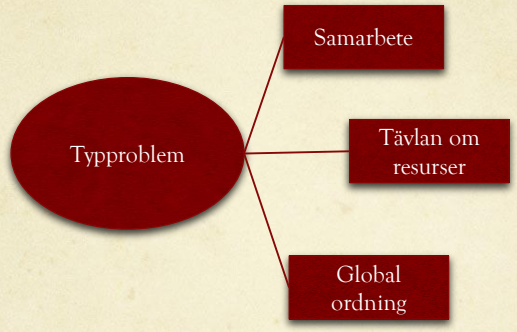
- Processtyper (s):
  - värd och gäst
- Godtyckligt #gäster
- Värden
  - fyller på förfriskningar i en bålskål
- Gästerna
  - dricker och umgås omväxlande
- Hur garantera att
  - gästerna inte dricker från en tom skål?
  - värden fyller på vid rätt tidpunkt?
  - ingen törstig gäst blir utan?

- Krav:
  - kontrollera #glas kvar (i bålen)
    - Kräver mutual exclusion
  - värden meddelas #glas == 0
    - Kräver synkronisering
  - gästerna väntar tills #glas == max
    - Kräver synkronisering

3

Översikt av klassiska problem

Klassiska problem: Designmönster



Känna igen problemen

5

Dining philosophers

1 processtyp

6

- Mutual exclusion
  - Process i kritisk sektion -> grannar ej i kritisk sektion
  - Alla process utan grannar i kritisk sektion->
    - kan befinna sig i den kritiska sektion
- Kritisk sektion =
  - delade resurserna

7

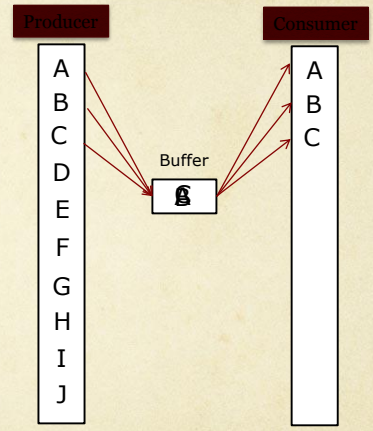
Drinking Philosophers

- Generalisering
- Godtyckligt nätverk
- Godtyckligt #grannar
- Om en granne i kritiska sektion ->
  - Process kan inte vara i kritisk sektion
- Global invariant (sann i **alla** möjliga tillstånd)
  - $\forall i,j(($ eating[i] $\wedge$ neighbor[i,j]) $\rightarrow(\neg$ eating[j]))

8

Producer/Consumer

- Två sorters processer
  - Delad buffer
  - Producenten lägger in data
    - (deposit)
  - Konsumenten hämtar data
    - (fetch)
- Överskrivning ej tillåten
- Om buffer uppdaterats ->
  - konsumenten får läsa
- Enkel buffer medför ->
  - P/C turas om



Global invariant – Producer/Consumer

- Mutual exclusion

$\forall i,j \neg(inDeposit[i] \wedge inFetch[j])$

- Producenten skriver inte över

$\forall i(inDeposit[i] \rightarrow counterProducer=counterConsumer)$

- Konsumenten läser inte samma värde två ggr

$\forall i(inFetch[i] \rightarrow counterConsumer<counterProducer)$

10

Producer/Consumer, forts.

- Cirkulär buffert med flera positioner
  - Samtidig åtkomst möjlig -> olika positioner hanteras
  - Antag att buffer[front] först och buffer[rear] slutet

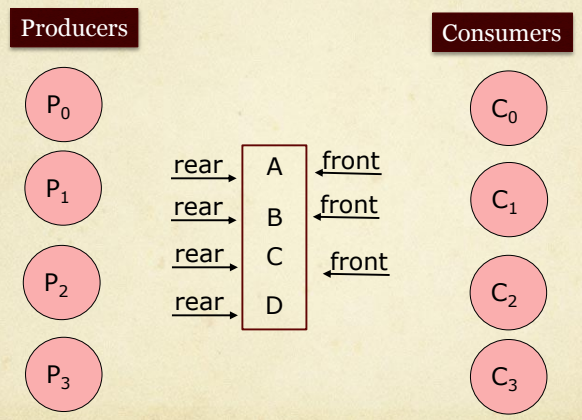
$\forall i,j(1 \le i,j \le n \mid (\text{inDeposit}[i] \wedge \text{inFetch}[j]) \rightarrow front \neq rear)$
$\forall i(\text{inDeposit}[i] \rightarrow \text{counterProducer} >= \text{counterConsumer})$
$\forall i(\text{inFetch}[i] \rightarrow \text{counterConsumer} < \text{counterProducer})$

- Godtyckligt antal processer

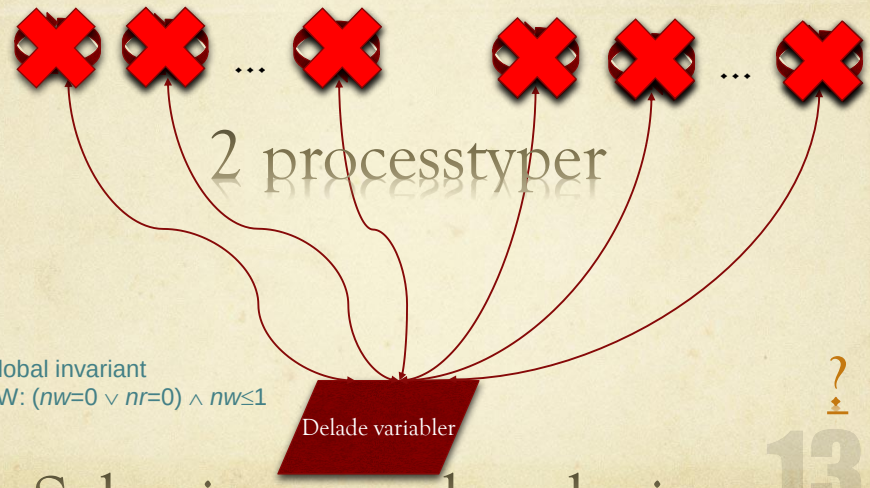
$\forall i,j(1 \le i,j \le n \mid (\text{inDeposit}[i] \rightarrow \neg \text{inDeposit}[j])$
$\forall i,j(1 \le i,j \le n \mid (\text{inFetch}[i] \rightarrow \neg \text{inFetch}[j]))$

11

Cirkulär buffer



Readers/Writers



Global invariant
RW: $(nw=0 \vee nr=0) \wedge nw \leq 1$

Delade variabler

Selective mutal exclusion

13

Dining Philosophers: Lösningsförslag

```
sem fork[n] := ([n]1)
process Philosopher(i := 1 to n)
do true →
  P(fork[i])
  P(fork[(i mod n)+1])
  eat
  V(fork[(i mod n)+1])
  V(fork[i])
  think
od
end
```

14

Dining Philosophers

```
sem fork[n] := ([n]1)
process Philosopher(i := 1 to n-1)
  do true →
    P(fork[i])
    P(fork[i+1])
    eat
    V(fork[i+1])
    V(fork[i])
    think
  od
end
```

```
process Philosopher(n)
  do true →
    P(fork[1])
    P(fork[n])
    eat
    V(fork[n])
    V(fork[1])
    think
  od
end
```

Dining Philosophers Java

```
public class PhilosopherLeftToRight implements Runnable {
   @Override
   public void run() {
      final int left=4;
      final int right=0;
      Random r=new Random(left);
      while (true) {
         GlobalProgramState.fork[left].P();
         GlobalProgramState.fork[right].P();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
         GlobalProgramState.fork[right].V();
         GlobalProgramState.fork[left].V();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
      }
   }

}
```

16

```
public class PhilosopherRightToLeft implements Runnable {
   @Override
   public void run() {
      final int left=AndrewsProcess.currentProcessId();
      final int right=left+1;
      Random r=new Random(left);
      while (true) {
         GlobalProgramState.fork[right].P();
         GlobalProgramState.fork[left].P();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
         GlobalProgramState.fork[left].V();
         GlobalProgramState.fork[right].V();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
      }
   }

}
```

17

```
public class GlobalProgramState {
   static AndrewsSemaphore fork[]=new AndrewsSemaphore[5];

   public static void main(String argv[]) {
      System.out.print(AndrewsProcess.licenseText());
      for (int i=0; i<GlobalProgramState.fork.length; ++i) {
         fork[i]=new AndrewsSemaphore(1);
      }
      RunnableSpecification rs[]=new RunnableSpecification[2];
      AndrewsProcess[] process;

      try {
         rs[0]=new RunnableSpecification(PhilosopherRightToLeft.class,4);
         rs[1]=new RunnableSpecification(PhilosopherLeftToRight.class,1);
         process = AndrewsProcess.andrewsProcessFactory(rs);
         AndrewsProcess.startAndrewsProcesses(process);
      } catch (InstantiationException | IllegalAccessException e) {
         e.printStackTrace();
      }
   }
}
```

18

Producer/Consumer: enkel buffer

```
sem full := 0
sem empty := 1
var buf : int

process Producer::
  var m : int
  do true →
    produce m
    P(empty)
    buf := m
    V(full)
  od
end
```

```
process Consumer::
  var n : int
  do true →
    P(full)
    n := buf
    V(empty)
    consume n
  od
end
```

Split binary semaphores

- Används
  - när processer delar kritisk sektion
  - men blockerar på **olika** villkor
- Invariant
  - $S_1 + S_2 + \ldots + S_n \leq 1$

Producer/Consumer: cirkulär buffer

```
sem full := 0
sem empty := N
var buf[N]: int
var rear : int := 1
var front : int := 1

process Producer::
  var m : int
  do true →
    produce m
    P(empty)
    buf[rear] := m
    rear:= rear mod N +1
    V(full)
  od
end
```

```
process Consumer::
  var n : int
    do true →
    P(full)
    n := buf[front]
    front := front mod N +1
    V(empty)
    consume n
  od
end
```

Producer/Consumer: Flera processer

```
sem full := 0
sem empty := N
sem mutexP := 1
sem mutexC := 1
var buf[N]: int
var front : int := 1
var rear : int := 1
```

```
process Producer(i:=1 to M)
   var m : int
   do true →
     produce m
     P(mutexP)
     P(empty)
     buf[rear] := m
     rear := rear mod N +1
     V(full)
     V(mutexP)
   od
end
```

```
process Consumer(j:=1 to M)
  var n : int
  do true →
     P(mutexC)
     P(full)
     n := buf[front]
          front := front mod N
+1
     V(empty)
     V(mutexC)
     consume n
  od
end
```

Producer/Consumer med cirkulär buffer i Java

```
public class Producer implements Runnable {
   @Override
   public void run() {
      int i=1;
      while(true) {
         GlobalProgramState.mutexP.P();
         GlobalProgramState.empty.P();
         GlobalProgramState.buffer[GlobalProgramState.rear]=i++;
         GlobalProgramState.rear=(GlobalProgramState.rear+1)%GlobalProgramState.n;
         GlobalProgramState.full.V();
         GlobalProgramState.mutexP.V();
         AndrewsProcess.uninterruptibleMinimumDelay(10);
      }
   }

}
```

23

```
public class Consumer implements Runnable {
   @Override
   public void run() {
      while(true) {
         GlobalProgramState.mutexC.P();
         GlobalProgramState.full.P();
         int value=GlobalProgramState.buffer[GlobalProgramState.front];
         GlobalProgramState.front=(GlobalProgramState.front+1)%GlobalProgramState.n;
         GlobalProgramState.empty.V();
         GlobalProgramState.mutexC.V();
         AndrewsProcess.uninterruptibleMinimumDelay(10);
      }
   }
}
```

24

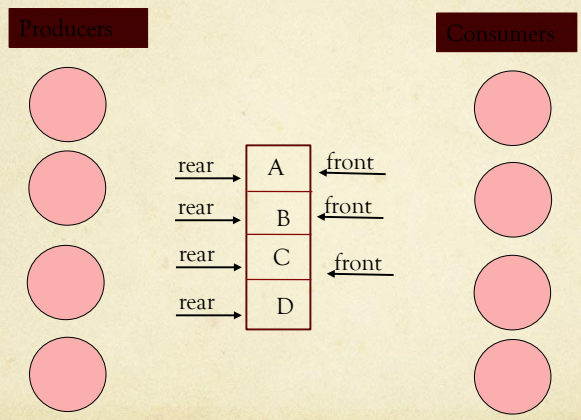
```
public class GlobalProgramState {
    public static int n=10;
    public static int buffer[]=new int[n];
    public static int front=0;
    public static int rear=0;
    public static AndrewsSemaphore empty=new AndrewsSemaphore(n);
    public static AndrewsSemaphore full=new AndrewsSemaphore(0);
    public static AndrewsSemaphore mutexP=new AndrewsSemaphore(1);
    public static AndrewsSemaphore mutexC=new AndrewsSemaphore(1);

    public static void main(String argv[]) {
        System.out.print(AndrewsProcess.licenseText());
        RunnableSpecification rs[]=new RunnableSpecification[2];
        rs[0]=new RunnableSpecification(Producer.class,10);
        rs[1]=new RunnableSpecification(Consumer.class,10);
        try {
            AndrewsProcess process[]=AndrewsProcess.andrewsProcessFactory(rs);
            AndrewsProcess.startAndrewsProcesses(process);
         catch (InstantiationException e) {
            e.printStackTrace();
         catch (IllegalAccessException e) {
            e.printStackTrace();
        }

    }
}
```

25

Cirkulär buffer



Readers/Writers

```
var nr, nw : int := 0
var dr, dw : int := 0
sem mutex := 1
sem read := 0
sem write := 0

process reader(i := 1 to N)
 do true →
     P(mutex)
     if (nw > 0) →
                    dr ++
        V(mutex)
        P(read)
     fi
     nr++
     signal()
     #do some reading
     P(mutex)
     nr- -
     signal()
     #do something else
  od
end
```

```
process writer(i := 1 to M)
 do true →
     P(mutex)
     if (nw > 0 or nr > 0) →
                    dw ++
        V(mutex)
        P(write)
     fi
     nw++
     signal()
     #do some writing
     P(mutex)
     nw- -
     signal()
     #do something else
 od
end

Procedure signal()
  if (nw = 0 and dr > 0) →
      dr - -
      V(read)
  [] (nr = 0  and nw = 0 and dw > 0) →
      dw- -
      V(write)
  [] (nw > 0 or dr = 0)  and (nr > 0 or nw > 0 or dw = 0) →
      V(mutex)
  fi
end
```

Readers/Writers i Java

```
public class Reader implements Runnable {
    @Override
    public void run() {
        Random r=new Random(AndrewsProcess.currentAndrewsProcessId());
        for (int i=0; i<GlobalProgramState.numberOfIterations; ++i) {
            GlobalProgramState.entry.P();
            if (GlobalProgramState.numberOfWriters>0) {
                ++GlobalProgramState.numberOfDelayedReaders;
                GlobalProgramState.entry.V();
                GlobalProgramState.delayedReader.P();
            }
            ++GlobalProgramState.numberOfReaders;
            GlobalProgramState.signal();

            AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
            GlobalProgramState.entry.P();
            --GlobalProgramState.numberOfReaders;
            GlobalProgramState.signal();
            AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
        }
    }
}
```

28

```
public class Writer implements Runnable {
   @Override
   public void run() {
      Random r=new Random(AndrewsProcess.currentAndrewsProcessId());
      for (int i=0; i<GlobalProgramState.numberOfIterations; ++i) {
         GlobalProgramState.entry.P();
         if (GlobalProgramState.numberOfWriters>0 || GlobalProgramState.numberOfReaders>0) {
            ++GlobalProgramState.numberOfDelayedWriters;
            GlobalProgramState.entry.V();
            GlobalProgramState.delayedWriter.P();
         }
         ++GlobalProgramState.numberOfWriters;
         GlobalProgramState.signal();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
         GlobalProgramState.entry.P();
         --GlobalProgramState.numberOfWriters;
         GlobalProgramState.signal();
         AndrewsProcess.uninterruptibleMinimumDelay(Math.abs(r.nextInt(1000)));
      }
   }
}
```

29

RW GlobalProgramState fragment

```
public static void signal() {
  if (numberOfWriters == 0 && numberOfDelayedReaders>0) {
     --numberOfDelayedReaders;
     delayedReader.V();
  } else if ( numberOfReaders == 0 &&
                numberOfWriters == 0   &&
                numberOfDelayedWriters>0) {
     --numberOfDelayedWriters;
     delayedWriter.V();
  } else {
     entry.V();
  }
}
```

30

```
public static AndrewsSemaphore entry = new AndrewsSemaphore(1);
public static AndrewsSemaphore delayedReader = new AndrewsSemaphore(0);
public static AndrewsSemaphore delayedWriter = new AndrewsSemaphore(0);
public static int numberOfWriters=0;
public static int numberOfReaders=0;
public static int numberOfDelayedReaders=0;
public static int numberOfDelayedWriters=0;
public static int numberOfIterations=100;
```

31

Partyproblemet

- Processtyper (s):
  - värd och gäst
- Godtyckligt #gäster
- Värden
  - fyller på förfriskningar i en bålskål
- Gästerna
  - dricker och umgås omväxlande
- Hur garantera att
  - gästerna inte dricker från en tom skål?
  - värden fyller på vid rätt tidpunkt?
  - ingen törstig gäst blir utan?

32

- Krav:
  - kontrollera #glas kvar (i bålen)
    - Kräver mutual exclusion
  - värden meddelas #glas == 0
    - Kräver synkronisering
  - gästerna väntar tills #glas == max
    - Kräver synkronisering

?

33