## *Process Synchronization*

**Synonym phrases**

- Process synchronization
- Process concurrency
- Process Parallelism

## *Intra-Concurrency*

**Precedence Graph**

**A directed acyclic graph whose nodes correspond to individual statements of a process and edges corresponds to the order in which statements are executed.**

## *Process Synchronization*

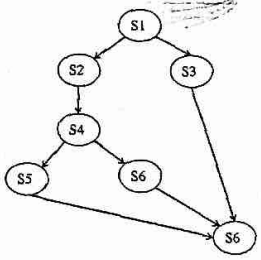
**Concurrency may happen in two levels:**

- **Intra-concurrency**
  - **(parallelism within a process)**
- **Inter-concurrency**
  - **(parallelism among several processes)**

## *Intra-Concurrency*

**A Simple Example**

```
a = 5;          S1: a = 5;
b = a*2;        S2: b = a*2;
c = a/2;        S3: c = a/2;
x = b+5;        S4: x = b+5;
y = x * 8;      S5: y = x * 8;
z = x - 4;      S6: z = x - 4;
g = c + y + z;  S7: g = c + y + z;
```




## *Intra-Concurrency*

**Goal:**

**Achieving parallelism within a process.**

**To meet the goal:**

- **Use Precedence graph which is implemented by the following constructs:**
  - **Fork . . . Join**
  - **Parbegin . . . parend**

## *Intra-Concurrency*

S7: g = c + y + z;

Variable whose value is changed (write into)
It makes the set:
W(S7) = {g}

Variables whose values are needed (read from)
They make the set:
R(S7) = {c, y, z}

## *Precedence graph*

**Concurrency condition** *(Bernstein Conditions)*

**Two statements of $S_i$ and $S_j$ can concurrently be executed if**

a. $R(S_i) \cap W(S_j) = \varnothing$

b. $W(S_i) \cap R(S_j) = \varnothing$

c. $W(S_i) \cap W(S_j) = \varnothing$

## *Fork & Join*

```
Fork L1
  •
  •
  •
  Go to L2
L1: •
  •
  •
L2: •
  •
  •
```

## *Implementation of a Precedence graph*

Use of

Fork . . . Join

Parbegin . . . parend

## *Fork & Join*

**Join is a construct that brings together the two parallel processes to continue with a sequential process.**

**<label> Join <counter>**

## *Fork & Join*

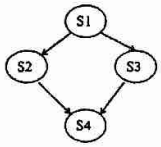
**Fork is a construct that indicates the beginning of two parallel executions**

**Fork <label>**

## *Fork & Join*

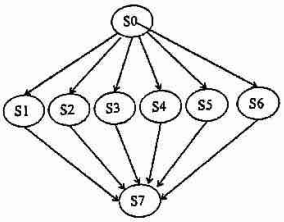
```
count =2;
Fork L1
    •
    •
    •
    Go to L2
L1  •
    •
    •
L2: Join count
```

## *Fork & Join*



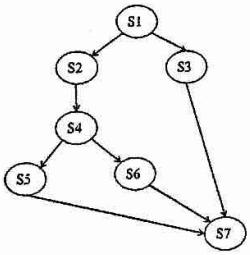

```
Count = 2;
S1;
Fork  L1;
   S2;
   Go to L2;
L1:  S3;
L2: Join count;
    S4;
```

## *Parbegin . . . parend*




$S_0$
Parbegin
$S_1$;
$S_2$;
$S_3$;
$S_4$;
$S_5$;
$S_6$;
parend
$S_7$;

## *Fork & Join*




```
Count = 3;
S1;
Fork  LS3;
      S2;
      S4;
   Fork LS6
      S5;
   Go to LS7;
   LS6:  S6;
   Go to LS7;
LS3:  S3;
LS7: Join Count
      S7;
```

## *Parbegin . . . parend*

Is *Parbegin . . . Parend* powerful enough to handle any precedence graph? NO.

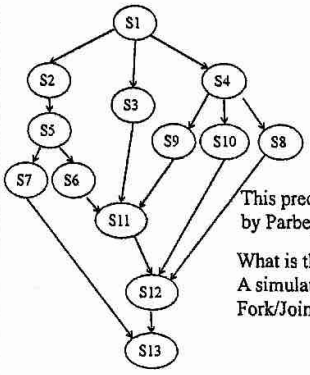
## *Parbegin . . . parend*

$S_0$

Parbegin $S_1, S_2, \ldots S_n$ parend

$S_{n+1}$

## *Example*




This precedence graph cannot be handled by Parbegin . . .Parend statement.

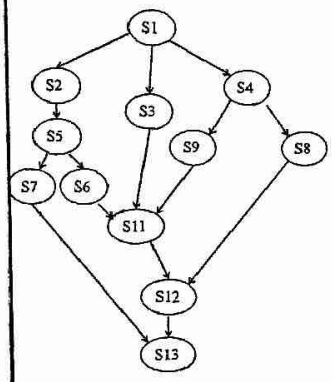
What is the solution?
A simulated concurrency statement using Fork/Join statements

## Back Ground

Parbegin $s_1, s_2, \ldots, s_n$ Parend statement could be simulated by Fork/Join as follows:

The given example cannot be handled by Parbegin $s_1, s_2, \ldots, s_n$ Parend. However, the simulated one can handle it but count=n will be replaced by other counters

```
Count = n;
Fork L2;
Fork L3;
   .
   .
   .
Fork Ln
   S1;
   Go to Lj ;
L2: S2;
Go to Lj;
L3: S3;
Go to Lj;
   .
   .
   .
Ln: Sn;
Lj: Join Count;
```




```
countS11=3; countS12 =2; countS13=2;
S1;
Fork LS2
Fork LS4
  S3;
  Go to LS11;
LS2: S2;
  S5;
 Fork LS6
  S7;
  Go to LS13;
LS6: S6;
  Go to LS11;
LS4: S4;
Fork LS9
  S8;
  Go to LS12;
LS9: S9;
LS11: Join countLS11;
  S11;
LS12: Join countLS12
  S12;
LS13: Join countLS1
S13;
```