Concurrency Divisions

1. Instruction level - executing 2 or more instructions at the same time

2. Statement level - executing 2 or more statements at the same time

3. Unit level - executing 2 or more subprograms at the same time

4. Program level - executing 2 or more programs at the same time

Categories of Concurrency

Quasi concurrency – Coroutine

Physical concurrency - Processor

Logical concurrency – Process, Task

Coroutines

1. All units have an equal relationship.
2. Control may be returned to the caller after partial execution.
3. Resume execution at the last statement executed on a previous call – multiple entries
4. Only one unit can execute at a time.
5. Single thread of control

Subprograms

1. A single entry point
2. Return control to the caller when execution terminates.
3. Suspend the calling program when they are executing.
4. Single thread of control

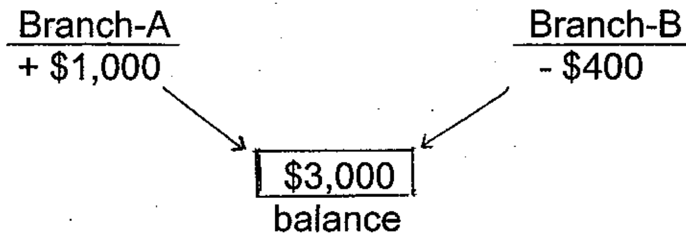
Concurrent Units

1. More than one unit can be executing at a time (either on a multiprocessor or logically using time-slicing)
2. Multiple threads of control

Concurrency

- Operating System

Motivation



Branch-A	Branch-B
load balance on ac	load balance on ac
add 1000 to ac	subtract 400 from ac
store ac on balance	store ac on balance

Scenario

	Branch-A	Branch-B
	load balance on ac (3000)	
		load balance on ac (3000)
	add 1000 to ac (4000)	subtract 400 from ac(2600)
1.	store ac on balance (4000)	
		store ac on balance (2600)
	or	
2.		store ac on balance (2600)
	store ac on balance (4000)	

Note. Race Condition

Fundamental Concepts

Tasks

Synchronization

- Cooperation

  Ex. Producer-Consumer Problem

Producer → buffer → Consumer

buffer

- Competition

  Ex. Task A: add 1 to Total
  Task B: multiply 2 to Total

Total: 3 - - - - - - - - - - - - 4 - - - 6 - - - -

Task A: Fetch Total → Add 1 → Store Total

Task B: Fetch Total → Multiply by 2 → Store Total

Question: How many different values of Total possible?

Liveness/Deadlock

Design Issues for Concurrency

How is competition synchronization provided?

How is cooperation synchronization provided?

How and when do tasks begin and end execution?

Are tasks statically or dynamically created?

Concurrency requires competition synchronization

Methods providing competition synchronization

Semaphores

Monitors

Message passing

P, V Semaphore

- Dijkstra (1965)

- P (proberen) - "wait"
- V (verhogen) – "signal"

```
(1)  P(S):  while S <= 0 do skip              // busy wait
            S:= S - 1;

     V(S):  S:= S + 1;                        // release
```

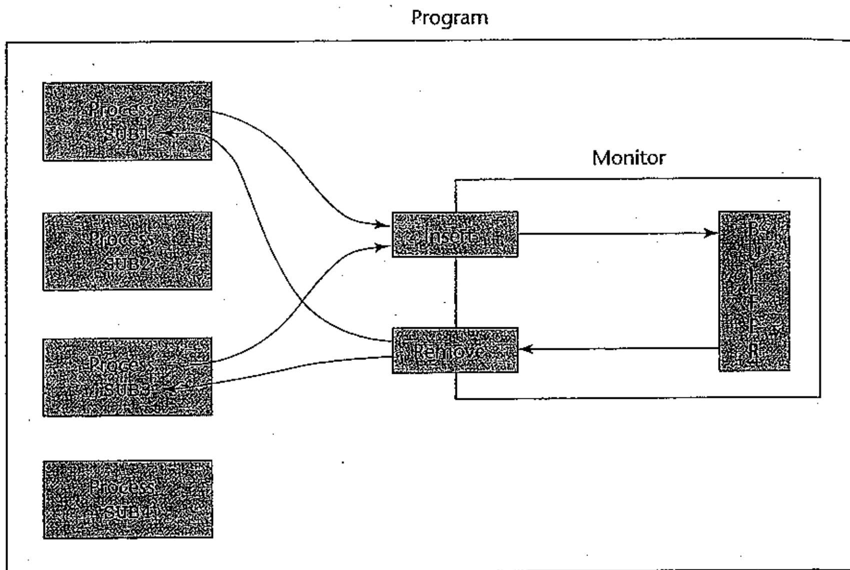
```
(2)  P(S):  S.value:= S.value – 1;
            if S.value < 0 then
               begin
                  add this process to S.L;    // S.L: queue for Semaphore
                  block;
               end

     V(S):  S.value:= S.value + 1;
            if S.value <= 0 then
               begin
                  remove a process P from S.L;
                  wakeup(P);
               end
```

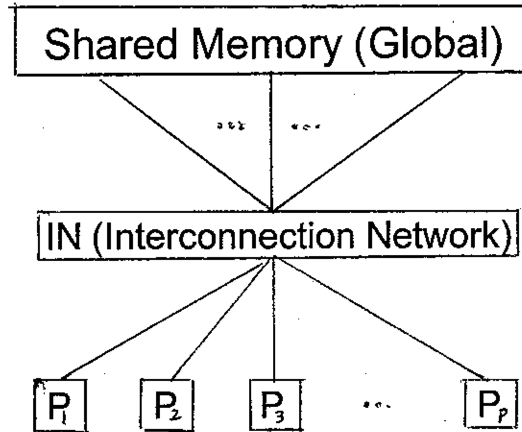
Monitor

- per Brinch Hansen
- Concurrent Pascal

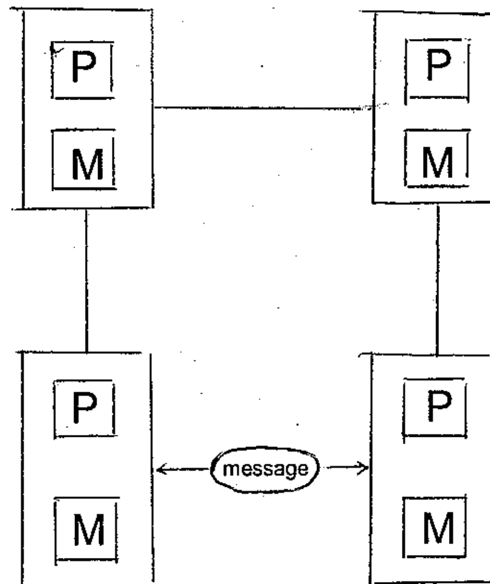
  Modula
  Ada
  Java
  C#



Shared Memory Architecture (Multiprocessor)



Distributed Memory Architecture (Multicomputer)



PVM / MPI

Parallel Programming Languages

Need:
1. Define a set of subtasks to be executed in parallel
2. Start and stop their execution
3. Coordinate and specify their instructions while they are executing.

Parallel Program Constructs

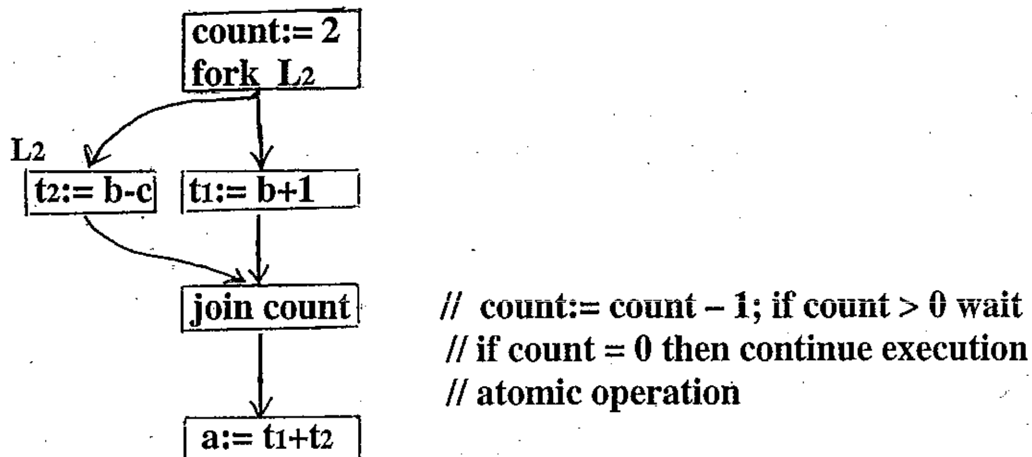
Thread – execution stream
Task – a procedure which can be executed with other procedures in parallel

(1) fork/join

Conway (1963)
Most primitive, flexible but not structured
PL/I, Unix

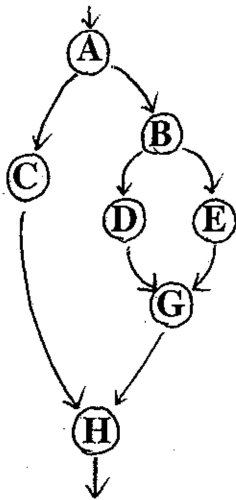
```
Fork L1      // new concurrent execution
Join  m,g    // m:= m -1; if (m = 0) then goto g
```

Ex. a:= (b + 1) * (b – c)



(2) parbegin/parend (cobegin/coend)

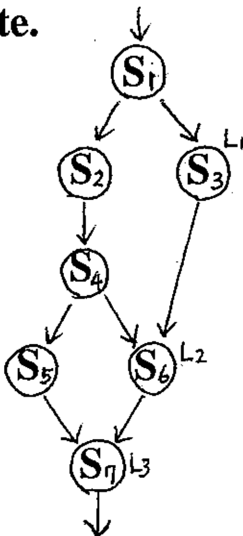
Dijkstra (1965)
High-level language construct
Degree of parallelism is static
Algol-68, CSP



precedence graph

```
A
parbegin
   C
   begin
      B
      parbegin
         D
         E
      parend
      G
   end
parend
H
```

Note.



fork/join

```
     S1
     count1 = 2
     fork L1
     S2
     S4
     count2 = 2
     fork L2
     S5
     goto L3
L1:  S3
L2:  join count1
     S6
L3:  join count2
     S7
```

parbegin/parend

?

(3) **forall (doall, pardo, doacross)**

degree of parallelism – dynamic
implemented with fork/join
vector and matrix operation

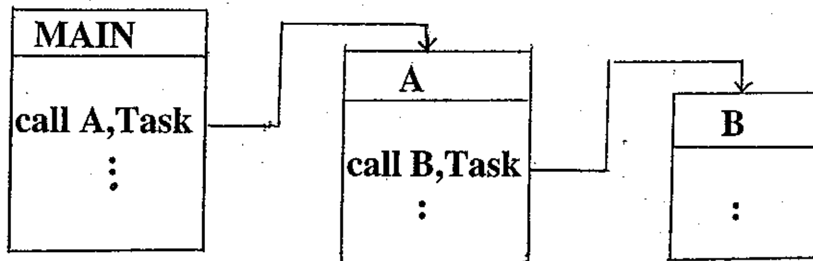
Fortran 95

Ex:
```
for i:= 1 to m do
  for j:= 1 to n do
    for k:= 1 to p do
      A[i,j,k]:= 0
    endfor
  endfor
endfor
```
→
```
forall (i=1:m, j=1:n, k=1:p)
    A(i,j,k) = 0
end forall
```
// construct

forall (i=1:m, j=1:m, k=1:p) A(i,j,k) = 0 // statement

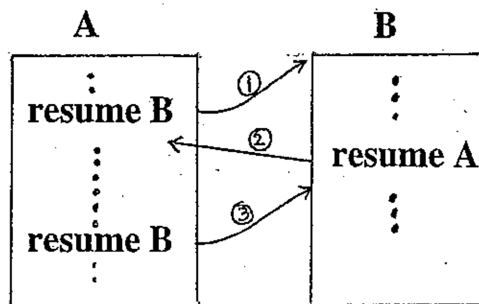
(4) **parallel subprogram**
declare processes, tasks, procedures, subroutines for parallel execution

Ex. PL/I Task - coordination is difficult.



Note. **Coroutine (Symmetric subprograms)**

Simula 67
Modula 2



Note: single thread

Concurrency in Programming Languages

(1) Shared-Memory

- o Algol 68: parbegin
  ---
  parend

- o PL/I: create TASK / terminate TASK

- o Concurrent Pascal
- o Lisp* for CM-2 (64K processor SIMD)
- o Ada
- o Fortran 90

(2) Message-Passing (Distributed Memory)

general form: SEND / RECEIVE

- o CSP (Concurrent Sequential Process)

```
parbegin
    send(!)
    receice(?)
```

- o Occam (for INMOS tansputer)

```
SEQ
    PAR
        channel1 ? partial1
        channel2 ? partial2
    sum:= partial1 + partial2
    channel3  ! sum
```

- o Ada

  Rendezvous

Statement-level Concurrency

HPF – extension of Fortran 90

- o Processors

  !HPF$ PROCESSORS procs(4) // 4 processors

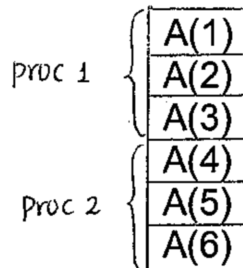
- o Data distribution

  !HPF$ DISTRIBUTE (kind) ONTO procs :: id_list

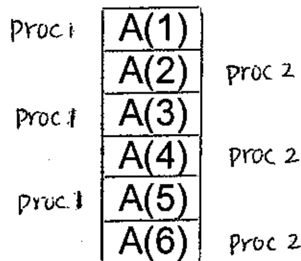
  kind is either BLOCK or CYCLIC

Ex. Integer A(6)
!HPF$ PROCESSORS procs (2)

(1) !HFP$ DISTRIBUTE (BLOCK) ONTO procs :: A



(2) !HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: A



- o Control Structure

  FORALL (I = 1:1000) A(I) = B(I)