

Chapter 2

Concurrent Programming

Introduction

Early systems:

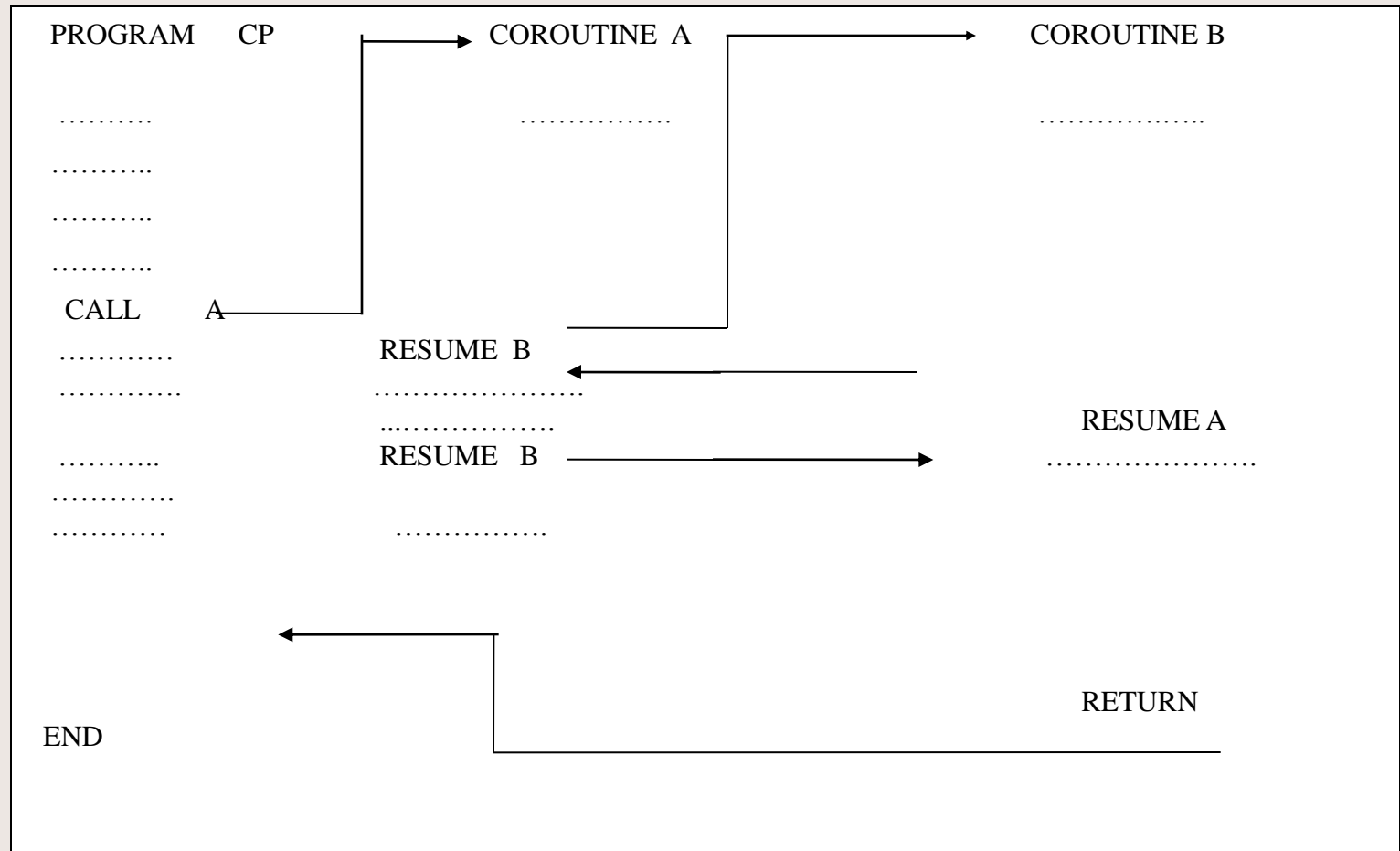
- 1) One program running but waiting for I/O caused loss of CPU cycle.
- 2) This problem was addressed & low level solutions were commonly implemented using interrupt I/O, giving overlapped CPU & I/O execution (CONCURRENCY OF CPU & I/O).
- 3) The motivation for concurrency specification to handle the programming tasks in such system was immediately felt.

Co routine:

A programming construct to specify the concurrency was born. Concept is quite general (not tied with I/O as such "CONWAY 1963:Design Of Separable Transition Diagram Compiler" CACM July 1963)

Co routine: Illustration

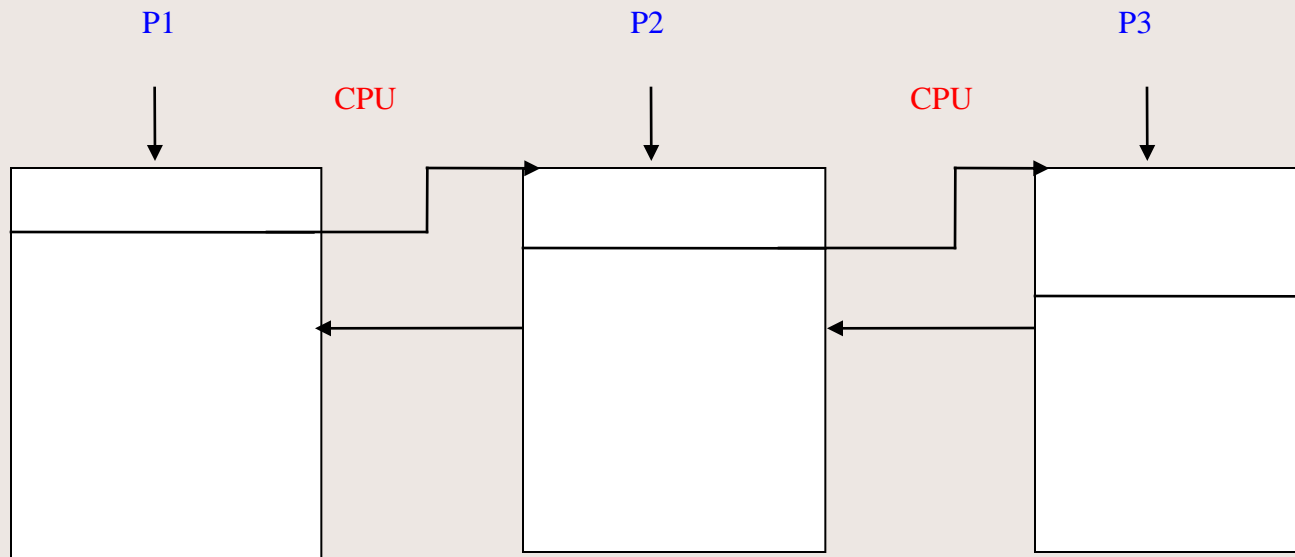
(Concurrency due to interleaved execution)



Coroutine (Cont)

1. Concept of coroutine was invented for the purpose of specifying interleaved execution.
2. Each coroutine, if viewed as a process , execution of resume causes process synchronization
3. When used with care coroutines are logically acceptable way of organizing concurrent programs,that share a single CPU .
4. In essence coroutines are concurrent processors in which process switching is completely specified statically by coroutine programmer rather than left to discretion at run time.
5. In 60's & 70's many languages(SIMULA 1,BLISS,MODULA) were developed to support concurrent programming using coroutine constructs.
6. CPU sharing between programs

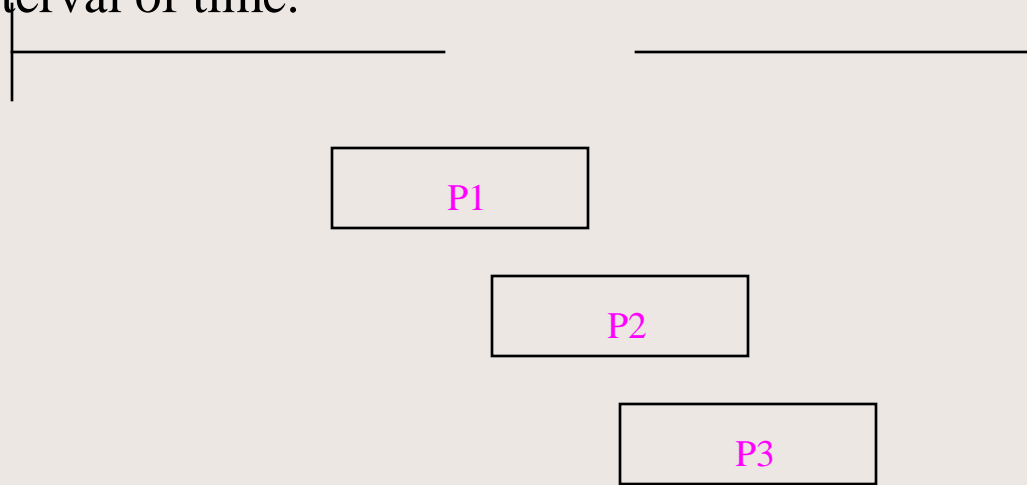
CPU sharing among n programs



1. Is this different from having n slower processors? At least in a logical sense!
2. The program $P_1, P_2 \dots P_n$ are executed concurrently.

Concurrent:(ISO definition)

Pertaining to the occurrence of 2 or more activities within a given interval of time.



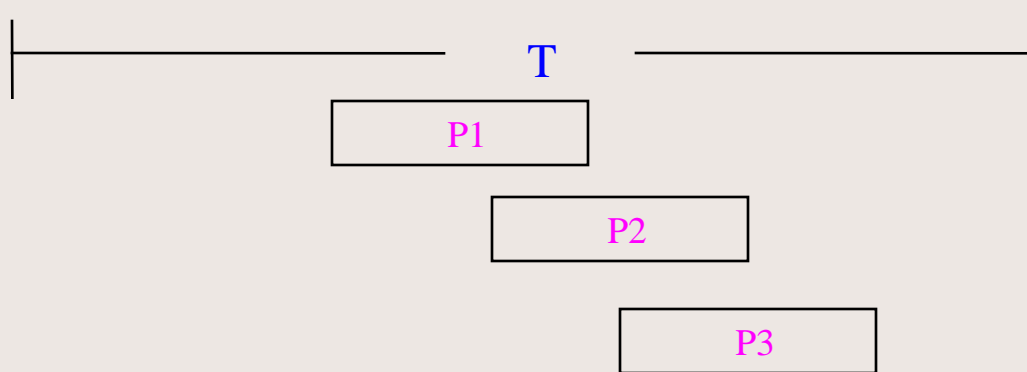
1. During an interval T ---P1 , P2 , P3 are said to be concurrent.
2. Here P1, P2 & P3 are processes or activities or programs.
3. Concurrent programs P1,P2 & P3 may shift in time (within the interval) And yet should produce the same result for an outside observer.

Concurrent:(ISO definition)

What we mean thus include

Simultaneously all the programs are being executed.

- 1.We need not deal with exact simultaneousness at a particular moment of time (rather we look at some non-zero interval of time)
- 2.Activities may be placed anywhere in T . Thus 2 or more programs are simultaneously (mainly concurrently) running if they run in interval T.
- 3.Overlap is not an essential feature.



P1, P2, P3 are concurrent during interval T but they do not overlap.

Examples of concurrency

1. A student may be doing 5 courses concurrently in this semester.
 2. Dr. XYZ is guiding 2 students, teaching 2 courses and is on 6 committees.
-

Concurrency is an abstract notion of parallelism

1. All of which may not manifest physically at all the times.
2. Physical manifestation of concurrency on n processors is denoted as n -fold parallelism.
3. Although parallelism was meant to denote actual physical parallelism, today it is also used in the sense of 'abstract parallelism'
4. Concurrency thus stands for a logical concept and it denotes abstract or logical parallelism.

Problems to be dealt with in concurrent Processes

- Specification
 - Communication
 - Synchronization
-

Language constructs for specification

Consider the following program segment

S1: Read (x);

S2: Read (y);

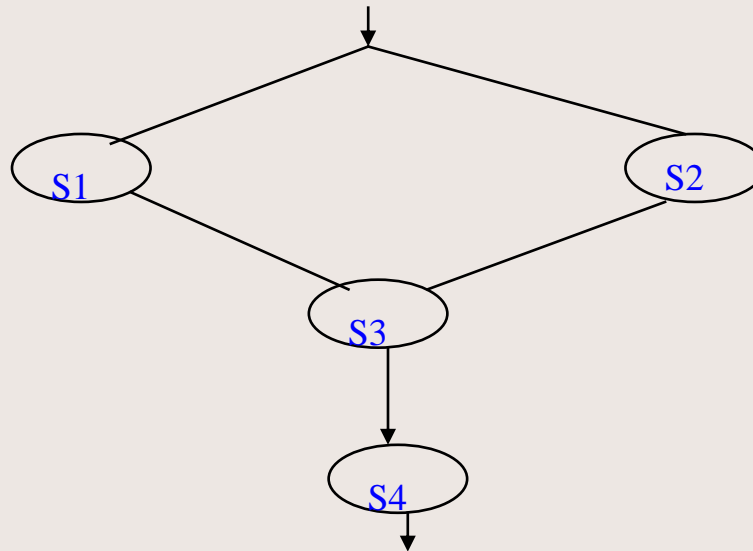
S3: $z := x + y$;

S4: Write (z);

Observations

1. The above sequence is written due to lack of constructs in a language
2. Statements S1 & S2 can be executed in parallel.
3. In fact the dependencies can be depicted by a precedence graph as

below.



When can two statements (consecutive) S_i , and S_j be executed concurrently?

1. Bernstein's conditions: Two consecutive statements S_i and S_j (S_j follows S_i) can be executed concurrently if all the followings hold:

1. $R(S_i) \cap W(S_j) = \{ \}$
2. $R(S_j) \cap W(S_i) = \{ \}$
3. $W(S_i) \cap W(S_j) = \{ \}$

Where $R(S)$ = domain of S

$W(S)$ = Range of S

\cap = Set intersection

$\{ \}$ = Empty set

Domain $R(S)$ of a statement S is a set of variable whose values are used by S (Read reference by S)

Range $W(S)$ of a statement S is a set of variable updated by the statement S .

For the above segment

$$W(S1)=\{x\}$$

$$R(S1)=\{ \}$$

$$W(S2)=\{y\}$$

$$R(S2)=\{ \}$$

$$W(S3)=\{z\}$$

$$R(S3)=\{x,y \}$$

$$W(S4)=\{ \}$$

$$R(S4)=\{z \}$$

From these set we observe that

$$1. \quad W(S1) \wedge R(S2) = \{ \}$$

$$W(S2) \wedge R(S1) = \{ \}$$

$$W(S1) \wedge W(S2) = \{ \}$$

➔ S1 and S2 can be executed in parallel(All conditions holds for S1 and S2)

$$2: \quad W(S2) \wedge R(S3) = \{y\}$$

$$W(S2) \wedge W(S2) = \{ \}$$

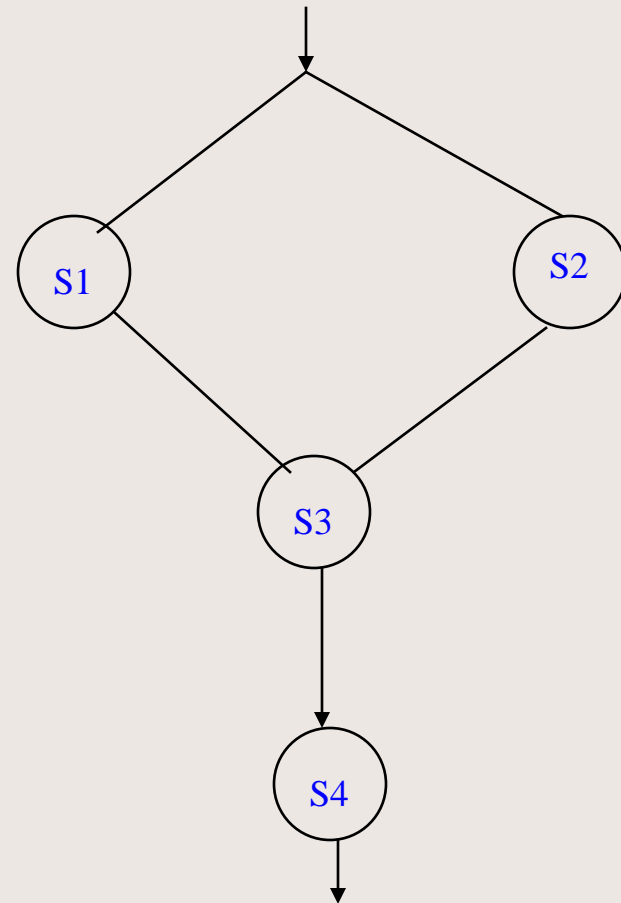
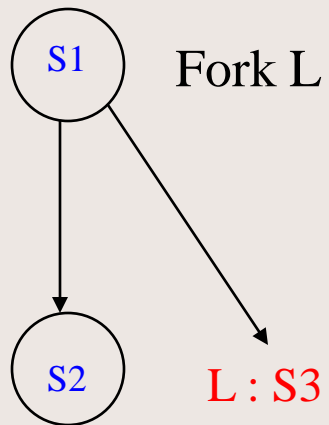
$$W(S2) \wedge W(S3) = \{ \} \quad \rightarrow \text{S2 and S3 can not be executed in parallel}$$

$$3: \quad W(S3) \wedge R(S4) = \{z\}$$

$$W(S3) \wedge W(S4) = \{ \}$$

$$W(S3) \wedge W(S4) = \{ \} \quad \rightarrow \text{S3 and S4 can not be executed in parallel.}$$

FORK and JOIN construct



FORK and JOIN construct

Fork L :

Creates a process at label L and another process at the statement followed by fork L.

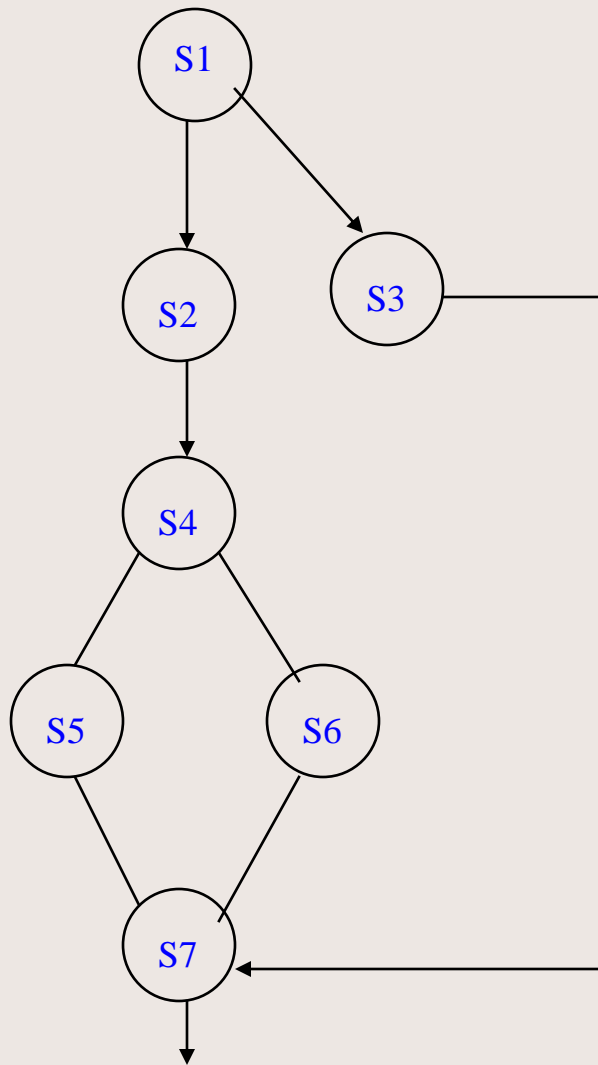
Join n :

1: Decrements n by one and keep the control if n is greater than zero.

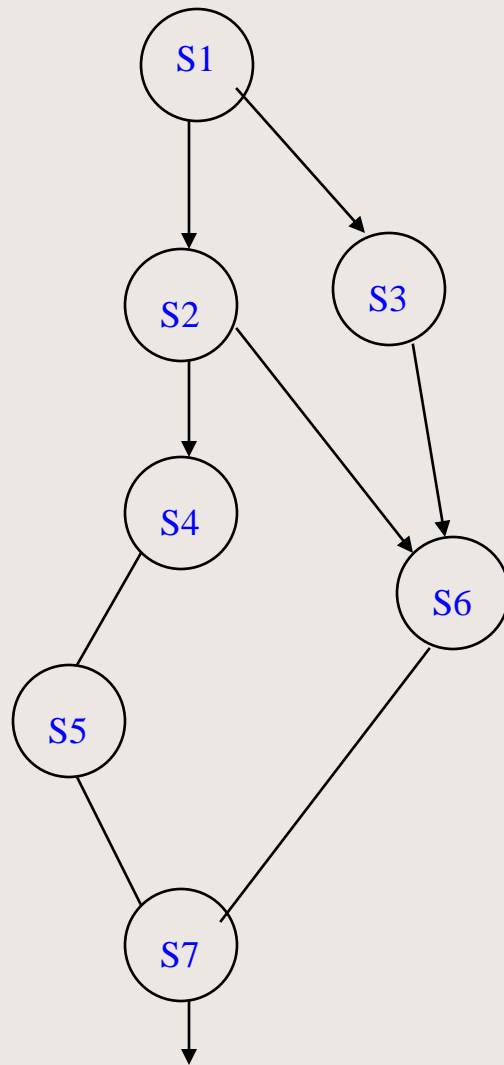
2: Join is atomic.

Atomic operation:

Concurrent execution has same effect as if it was executed sequentially by many in any arbitrary order.

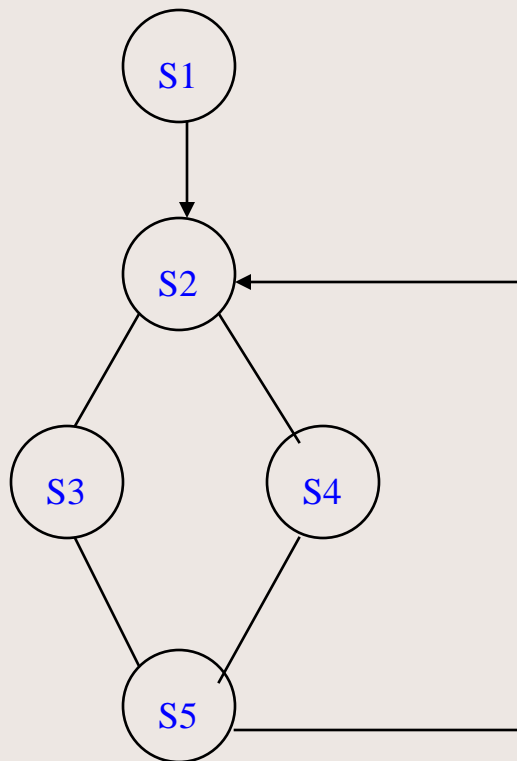


	Count =3;
	S1;
	Fork L1;
	S2;
	S4;
	Fork L2;
	S5;
	goto L3;
L2:	S6;
	goto L3;
L1:	S3;
L3:	Join Count;
	S7;



S1;
Count+1=2;
Fork L1;
S2;
Count+2=2;
Fork L2;
S4;
S5;
goto L3;
L1: S3;
L2: Join+Count1;
S6;
L3: Join + Count2;
S7;

Concurrent Program to copy a file onto another



Var f,g: file of T;

r,s:T;

Count: integer;

Begin

reset(f);

rewrite(g);

read(f,r);

While not eof(f) do

Begin

count:=2;

s:=r;

fork L1;

write(g,s);

goto L2;

L1: read (f,r);

L2: Join Count;

End;

Write(g,f);

End;

S1

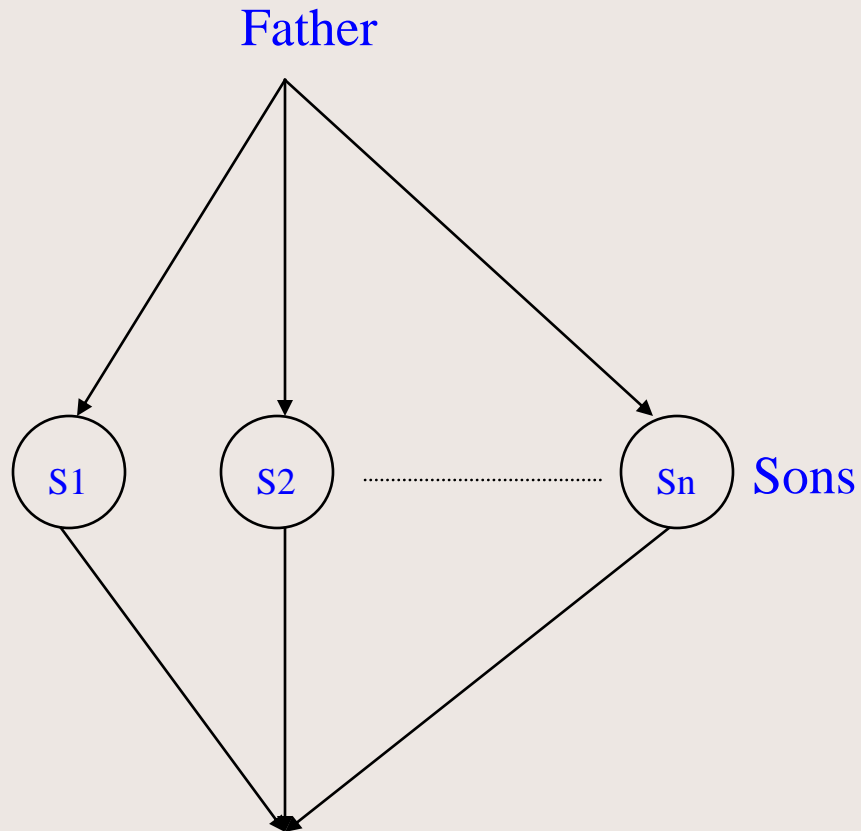
S2

S3

S4

S5

PARABEGIN- PAREND Constructs



Parbegin

S1;

S2;

.

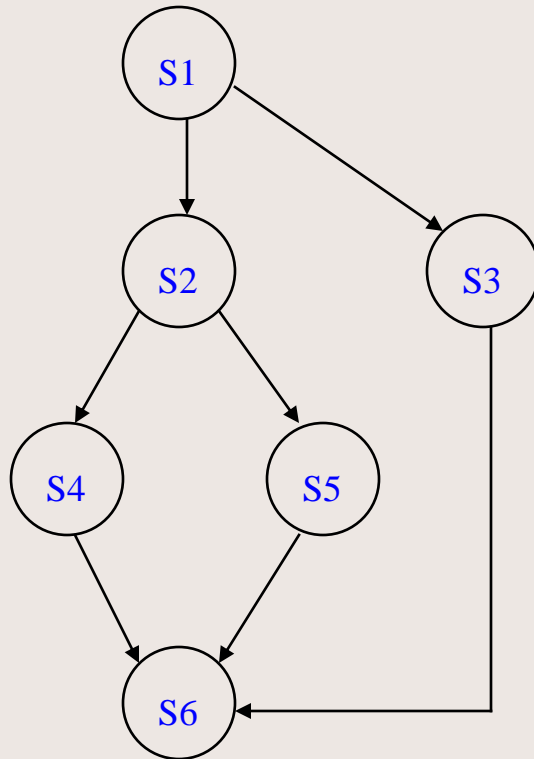
.

.

Sn;

Parend;

Example



S1; ↓
Parbegin

begin

S2; ↓
Parbegin

S4; →

S5; →

Parend

end;

S3; ↓

Parend;
↓
S6;

File Copy Program

Var f,g: file of T

r,s: T;

begin

reset(f);

rewrite(g);

read(f, r);

while not eof(f) do

begin

s:=r;

parbegin

write(g, s);

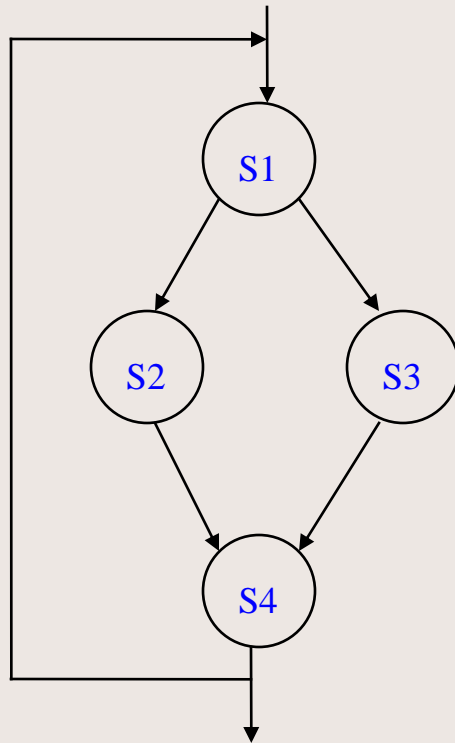
read(f, r);

parend;

end;

write(g, r);

end;

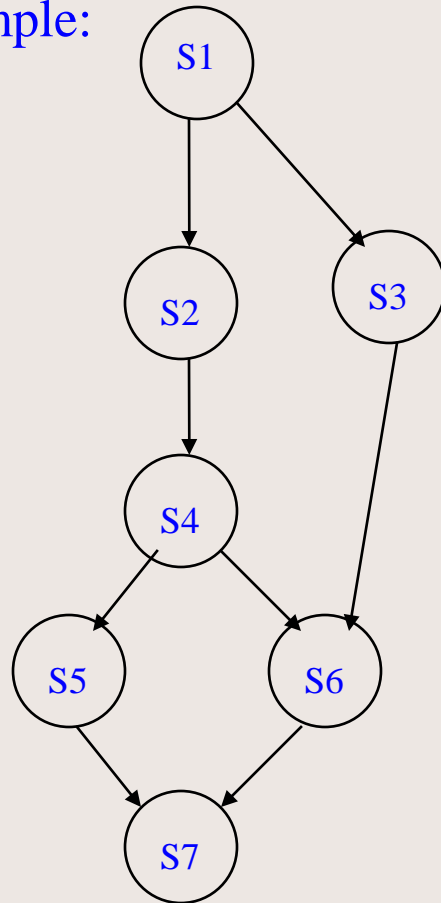


Exercise: Simulate parbegin.. parend using Fork...Join constructs.

Q) Is parbegin ... parend functionally complete? No

Note: Every precedence graph computation cannot be specified using parbegin... parend construct.

Example:



	S1;
	Count1:=2;
	Fork L1;
	S2;
	S4;
	Count2:=2;
	Fork L2;
	S5;
	goto L3;
L1:	S3;
L2:	Join Count1
	S6;
L3:	Join Count2;
	S7;

Comments on Fork and Join

1. Fork does process (or thread) creation and join does process destruction.
2. Not very good operations. Forks make programs unreadable due to go to use.
3. They create very clumsy programs interwoven by forks, go to and join.
4. Hard to debug and understand.
5. It is convenient to view a node of precedence graph as a process.
6. In such an environment processes appear and disappear dynamically during the lifetime of a single program execution.
7. This scheme may result in a significant overhead in terms of a number of process to be created and destroyed.
8. The overheads may be reduced if we collapse those activities that can be carried out sequentially into a single process.
9. This change need to be done cautiously so that we don't reduce the amount of concurrency allowed in the computation.
10. For example S2 and S4 in the precedence graph G1 can be combined into a single node.

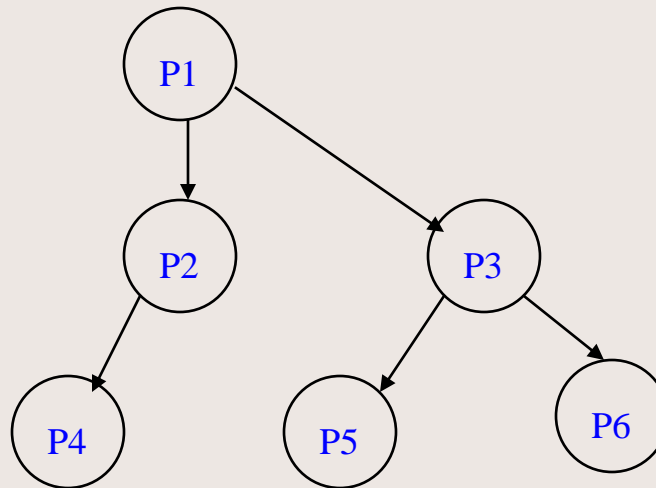
A process graph:[Family tree of processes]

1. Is a directed rooted tree, an edge from a process P_i to P_j means P_i has created P_j . Batch formation time,
 2. We call P_i as father and P_j as son.
 3. The graph must be a rooted tree(since a process can have only one father but it can have as many sons as it creates)
-

Process creation

A process creates another process by CREATE operation such as FORK.

Process Graph:



Several implementations are possible

1. Execution : Concurrent Vs Sequential

A: father continues to execute concurrently with sons

B: father waits until all sons have terminated.

2. Sharing : All Vs Partial

A: father and sons share all variables in common.

B: sons share only a subset of their father's variables.

Fork-join Semantics

OPTION 1.A has been adopted in fork-join.

When a process P_i executes FORK L a new process P_j is created (P_i is father, P_j is son) Both executes concurrently.

OPTION 2.A has been adopted in fork-join (Since all variables are shared)

OPTION 2.B has been adopted in UNIX (independent mem. Only files can be shared)

Par begin-parend-semantics

OPTION 1.B is adopted

Parbegin

S1: ➔ child Process executes

S2: concurrently and father

S4: waits till all terminates.

Parend

➔ father waits here.

Process termination

(a) Natural death(Termination) when executes last statements.

(b) A process can be killed by another by ABORT statement process with id is aborted (Kill process -id or ABORT process-id)

Abort operation can usually be invoked by father process.

Note that father needs to know ids of it's sons.

A father may end the life of one of its sons due to

(a):Son has exceeded the use of some resources it was allocated.

(b):Task assigned to son is no longer required.

Many systems don't allow sons to live if their father is terminated.

Many systems don't allow sons to live if their father is terminated

- OS must maintain proper information so that an orphan does not exist in the system. (Orphan process, difficult in distributed OS , as process graph is distributed)

CRITICAL SECTION PROBLEM

○ Banking System

Balance = 2000/-

P1 → deposits 3000/-

P2 → deposits 2000/-

T_0 : P1 reads B in R

T_1 : P1 adds 3000/- to R → P1 is interrupted

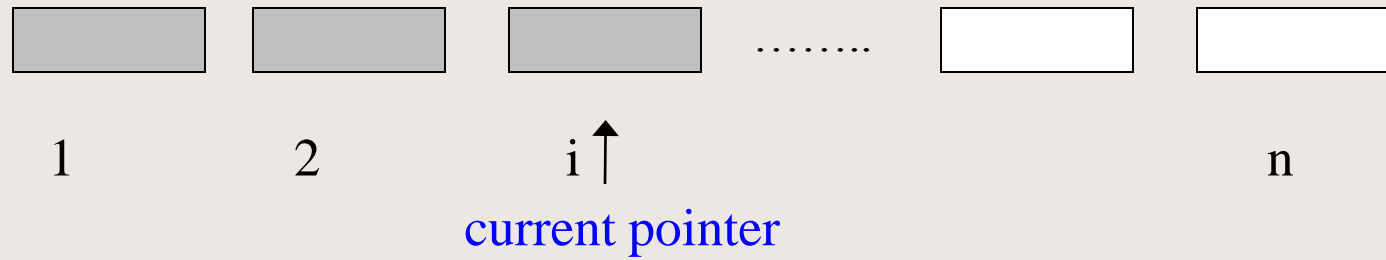
T_2 : P2 reads B in R

T_3 : P2 adds 2000/- to R

T_4 : P2 stores R in B → P2 is interrupted

T_5 : P1 stores R in B, Final balance = 5000/-
instead of 9000/-

Bounded Buffer Producer – Consumer Problem



- Buffer of n items ($1 \dots n$)
- Producer puts an item in the next place (after i)
- Consumer consumes an item from buffer pointed by I
- $i \rightarrow$ points to highest buffer that is filled

Producer :Repeat

while $i = n$ do NOP;

$i := i + 1$

produce(next P)

buffer[i] := next P;

until false

until false

consumer :Repeat

while $i = 0$ do NOP;

next C := buffer[i]

Consume(next C)

$i = i - 1$;

until false

- Producer – Consumer operates concurrently at their own speed.
- Possible sequence of events

T_0 : Producer executes
 while $i = n$ do NOP;
 $i := i + 1$;
 Produce(next P) ; Interrupted

T_1 : Consumer executes
 While $I = 0$ do NOP;
 Next C := buffer[I];
 Consume the next C;
 $I = I - 1$; Interrupted

T_2 : Producer resumes its operation

Result : Consumer consumes an empty item.
 Producer overwrites an item.

OBSERVATIONS

1. As sequential program produces consumer work correctly
2. They take care of boundary conditions (Buffer full or empty)
3. In concurrent mode, they do not work correctly
4. Certain parts of codes of processes are critical
5. We will get the correct result if it is guaranteed that only one process executes in the CS i.e. CS must be mutually exclusive
6. CS stands for indivisible operation

DIJKSTRA'S CONDITIONS

1. No assumptions to be made concerning hardware instructions or number of processors it supports generality of m/c
2. No assumptions on relative speed of process execution → generality of asynchronous execution
3. When a process is not in its CS it should not prevent others from entering their CS → mutual block
4. The decision must be made to admit the process into the CS in finite duration of time → prevents starvation

SOFTWARE APPROACH TO CS PROBLEM

Algorithm 1

var turn = 0;	→ turn common variable
While turn \neq i do NOP	→ initialized to 0/1
cs	→ mutual blocking
turn = j;	→ strict alternation

Algorithm 2

var flag : array[0..1] of Boolean	→ initialized to false
flag[i] = true	→ P _i is inside
while flag[j] do NOP;	
flag[i] := true;	→ system crash or indefinite wait
[CS]	→ simultaneous execution of P ₀ &
P1	
flag[i] := false;	

Algorithm 3

meaning of $\text{flag}[i] = \text{true}$

$\text{flag}[i] = \text{true};$ \rightarrow I want to enter

while $\text{flag}[j]$ do NOP \rightarrow If you want, I wait

CS

$\text{flag}[i] := \text{false};$ \rightarrow while leaving put down my flag

- Both may want to enter simultaneously and set their flags
- Both P0, P1 wait in while Loop \rightarrow Deadlock
- Mutual blocking constraint is violated

Algorithm 4

flag[i] := true → mutual blocking may arise

while flag[I] do

begin → both raising and lowering flags up & down indefinitely,

but rare

flag[i] := false;

while flag[i] do NOP;

flag[i] := true;

end;

[CS]

flag[i] := false;

DEKKER'S SOLUTION (FOR TWO PROCESSES)

var flag : array[0...1] of boolean;

turn : 0...1

initially \rightarrow flags are false

turn 0 or 1

flag[i] := true

\rightarrow I want to enter

while flag[j] do \rightarrow If you also want & your turn, I say I don't want and I wait till my turn

if turn = j then

begin

flag[i] := false;

while turn = j do NOP


flag[i] := true

end

[CS]

turn := j

flag[I] := false

- 
-
- ❖ Mutual Exclusive
 - ❖ No Deadlock
 - ❖ No mutual Blocking
 - ❖ Any process can enter in finite duration of time

Dijkstra's Solution (n-processes)

Var flag : array[0...n-1] of (idle, want-in, in-CS)

turn : 0...n-1

- Initially all flags are idle and turn is any value between 0 to n-1
- Process P_i

Var j : integer;

Repeat

flag[i] := wantin

while turn \neq i do

if flag[turn] = idle then

turn := i;

flag[i] := in-CS;

j := 0;

while (j < n) and (j = i or flag[j] \neq in-CS)

do j := j+1

until j \leq n;

[CS]

flag[i] := idle;

1.Mutual Exclusion

2.Mutual Blocking cannot occur

3.Does not ensure fairness → Indefinite wait, Starvation

Constraint 5

There must be a bound on the number of times that other processes are allowed to enter after a process has made its desire to enter.

Lamport Bakery Algorithm

1. Based on bakery shop, each customer receives a token
2. Allow each to be served in turn
3. Common data structures:
 - choosing [n] initially false
 - number [n] int, initialized to 0
4. Semantics:
 1. $(X,Y) < (R,S)$ if $X < R$ or $(X=R \ \& \ Y < S)$
 2. getmax(number [], n) returns an integer K s.t.
 $K \geq \text{number}[i], i=1,2,3,\dots,n$

Lamport Bakery Algorithm (Cont)

```
boolean choosing [n];
int number [n];
while (true)
{ choosing[I] =true;
number[I]= getmax(number[ ],n) +1;
choosing[I]=false;
for (int j=0;j<>n;j++)
{ while(choosing[j])      { };
  while ((number[j] !=0) && (number ([j] ,j) <> number ([i] ,i) { };
}
Critical Section
number[i]=0;
}
```

Lamport Bakery Algorithm (Cont)

1. Enter in FCFS
2. Mutually Exclusive
3. Deadlock free
4. Hardware solutions

Hardware Solutions

Atomic Operation:

1. Indivisible \rightarrow either full or none
2. If more than one process executes, they execute in any unspecified sequence
3. Join operation is atomic
4. Atomic operations are implemented in hardware and hence can be used in critical sections

❖ Test and set instruction (IBM 370)

Function test_and_set(var x:boolean):boolean;

Begin

test_and_set:=x;

X:=true;

End;

❖ If the M/C support test_and_set instruction as atomic instruction, then it can be used to solve CS problem

❖ While test_and_set(lock) do NOP;

CS

lock:=false

- ✓ lock is a common variable shared by all processes
- ✓ hardware lock
- ✓ if more than one processes execute test_and_set, one among them is going to execute it(guaranteed by hardware)
- ✓ Hence only one process enters
- ✓ Fairness is governed by the hardware which may have its own policy
- ✓ Mutual blocking not possible as unlocking is done at exit code, and only a process which enters has lock = true.

SWAP Instruction

Procedure swap (var a,b:boolean);

Var temp:boolean

Begin

temp:=a;

a:=b;

b:=temp;

End;

- ✓ A global variable lock initialized to false can be used by each process
- ✓ Each process has a local variable key. When a process wants to enter its CS it employs its key by setting it true and waits for the lock to become true.

Process P_i 's code looks like

key:=true;

Repeat

swap (lock,key);

until key=false;

CS

lock:=false;

- ✓ If a process finds lock = false and it wants to enter, its key is set to true
- ✓ If there are more than one processes executing the entry code, swap being atomic, one of the processes will make lock = true and its key = false, while others will swap their true values of the keys with true value of the lock and will be in the repeat loop.

- ❖ Integer variable in which two atomic operations P and V is defined to provide synchronization and mutual exclusion in concurrent systems (

◆ Definition :

P (n): L: if $n \leq 0$ then go to L ➔ wait

$$V(n): \quad n := n + 1;$$

Implementation of Semaphores in CS problem

- Processes share a common variable semaphore defined as

VAR mutex: semaphore:=1;

- Process P_i has the following organization

P (mutex) ➔ wait if somebody inside

CS

V (mutex) ➔ signals the finish

- Mutex can be set to K, if K processes to be allowed
- In CS normally, mutex=1

Semaphore for Synchronization

Var Sync:semaphore:=0;

Parbegin

begin S1;V(sync); End;

Begin P(sync); S2; End;

Parend;

Example:

Var a,b,c,d,e,f,g:semaphore:=0;

Begin

Parbegin

Begin S1;V(a); V(b);end;

Begin P(a);S3;S4;V(c);V(f);end;

Begin P(b);S2;V(d);end;

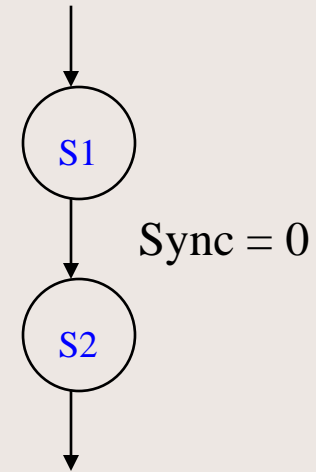
Begin P(c);S5;V(g);end;

Begin P(f);P(d);S6;V(e);end;

Begin P(g);P(e)S7;end;

Parend;

End;



Implementation of Semaphores

- ❖ Busy waiting
- ❖ Modify the P operation to block the process itself if semaphore is not positive, thus control could be given to CPU scheduler to schedule another process
- ❖ A blocked process on P operation can be waked up by V operation by another process, and changes its state from blocked to ready.

New Semaphore definition:

Type semaphore = record

Val: integer;

L: list of processes

End;

Var n: semaphore;

❖ P(n): $n.val := n.val - 1;$

 If $n.val < 0$ then

 Begin

 Add this process to n.L

 Block;

 End;

❖ V(n): $n.val := n.val + 1;$

 If $n.val \leq 0$ then

 Begin

 Remove a process P from n.L

 Wakeup (P);

 End;

- ✓ block operation suspends a process that invokes it
- ✓ wakeup operation resumes the execution of one of the process
- ✓ These operations are provided by kernel
- ✓ List of processes can be implemented by linked list of PCBs with FIFO principle
- ✓ Semaphore operation does not specify the mechanism by which a process is selected
- ✓ In uni-processors P , V can be implemented using interrupts(masking)
- ✓ In MP systems test_and_set or swap could be used

PRODUCER-CONSUMER Problem(bounded buffer)

Type

item=.....

Var

full,empty,mutex:semaphore;

nextp,nextc:item;

begin

 full:=0;

 empty:=n;

 mutex:=1;

 Parbegin

 Producer: repeat

 produce an item in nextp;

 P(empty);

 P(mutex);

 V(mutex);

 V(full);

 Until false;

PRODUCER-CONSUMER Problem(bounded buffer) (Cont)

Consumer: repeat

 P(full);

 P(mutex);

 remove an item from buffer to nextc

 V(mutex);

 V(empty);

Until false;

 Parend;

End;

READERS / WRITERS PROBLEM

Problem-1

No reader should be kept waiting unless a writer already has been allowed. (No reader waits because of waiting writer).

Problem-2

A ready writer (waiting) should be allowed to enter as soon as possible, i.e., if a writer is ready to enter, no new readers should be allowed.

READERS / WRITERS PROBLEM

Solution : Problem-1 :

- ✓ More than one reader inside
- ✓ Only one writer inside (no reader, no writer)
- ✓ Writers may starve
- ✓ Shared data structure
- Var mutex, wrt : semaphore := 1
- Readcount : integer := 0

Algorithm Principle :

- ✓ First reader check, no writer inside
- ✓ Other readers can simply enter (All readers increment the readcount)
- ✓ Every reader decrement the readcount by 1 while leaving
- ✓ Last reader should signal the writer

READER routine :

P(mutex);

Readcount := readcount + 1

If readcount = 1 then P(wrt);

V(mutex);

reading the data

P(mutex);

Readcount = readcount - 1;

If readcount = 0 then V(wrt);

V(mutex);

WRITER routine :

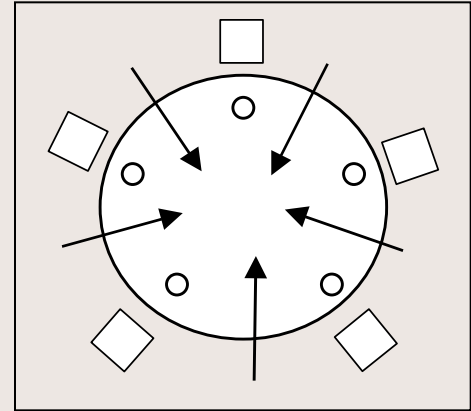
P(wrt);

writing is done

V(wrt);

DINING PHILOSOPHERS' PROBLEM

- Five philosophers in a common table
- Five chopsticks
- Philosophers think and eat
- Pickup one chopstick at a time
- When he has both the chopsticks he eats
- After eating, put down sticks and goes to think



SOLUTION

- ❖ Represent each chopstick by a semaphore
- ❖ Philosopher tries to grab a chopstick
- ❖ Release chopstick by V operation

Algorithm :

Var chopstick : array[0..4] of semaphore; := 1;

Philosopher process P_i

Repeat

 P(chopstick[I]);

 P(chopstick[I+1 mod 5])

 .

 eat

 V(chopstick[I]);

 V(chopstick[I+1]);

 .

 think

 .

until false

Observations :

- Guarantees no chopsticks are shared
- Deadlock possibility
 1. All five become hungry simultaneously
 2. Executes P operation
 3. Each get one and never get the second
- Remedy
 1. Allow at most four to be sitting (at least one should be prevented from eating)
 2. Allow a philosopher to pickup his both chopsticks if both are available
 3. Use asymmetry → odd philosopher picks up left
 4. Even philosopher picks up right first
- Any solution is deadlock free, but starvation may occur

CONCURRENT PROGRAMMING

- Operating systems are nowadays almost written in higher level languages.
- It improves implementation, maintenance and portability.
- Language constructs that support concurrent programming.
- Semaphore is one of such constructs useful in concurrent programming.
- Semaphore is a very low level constructs and its use gives rise to some problems as we have with gotos in sequential languages.
- What is needed is structured programming constructs.

Bad readability of semaphores

A single inappropriate use due to a programming error(i.e. a single process misbehaving could result in great difficulties.)

A critical section like below

V(mutex);	P(mutex)	
:		.
critical section		CS
:		:
P (mutex)	P(mutex)	→error & cause

Deadlock

- Or if one of the operations is omitted then either deadlock will occur or mutual exclusion is violated.
- Time dependent errors can thus be easily generated when semaphores are used incorrectly by even a single process. This process will make other processes also do wrong !

Bad readability of semaphores

- Region Construct, by CAR Hoare [1972] and Hansen [1972].
- A variable V of type T that is required to be shared among many processes can be declared as:

Var V : shared T;

Region V do S; ----Region construct

- It means that when S is being executed no other process can access shared variable V.

i.e. Region V do S1

 Region V do S2

- If are executed concurrently the effect will be sequential execution
S1 followed by S2 or
S2 followed by S1

Bad readability of semaphores

- Var free : shared array [1..n] of Boolean;

Procedure acquire (var index : integer);

Begin

 Region free

 do for index := 1 to n do

 if free [index] then

 begin

 free [index] := false;

 exit;

 end;

 index := -1;

end;

Procedure release (index : integer);

begin

Region free do

begin

free [index]:= True;

end;

end;

Implementation of critical regions

- (Compiler supporting region statement in a programming language will use semaphores internally).

Region V do S

compiler generates a semaphore

V-mutex : semaphore := 1;

The code shall be same as that for critical section:

P(V-mutex); S; V(V-mutex)

- Regions are critical sections and when a higher level language supports it, nesting of regions could create problems.

Implementation of critical regions

Example

Var X, Y : shared T;

Parbegin

Q: Region X do region Y do S1;

R: Region Y do region X do S2;

Parend;

If Q, R enter regions X, Y respectively at about the same time, a dead lock may occur.

Implementation of critical regions

- Consider the time sequence & actions as:

T0: Q executes P(x_mutex)

T1: R executes P(y_mutex)

T2: R executes P(x_mutex); R waits since x_mutex = 0;

T3: Q executes P(y_mutex);); Q waits since y_mutex = 0;

Both are waiting for a signal from each other.

- Deadlock occurs because regions are nested in each other.
- Compilers can detect such a possibility and error could be given indicating deadlock possibility as follows:
- If we have two regions x & y and if y is nested in x, it is denoted by $y < x$. It will find also $x < y$ if mutually nested. Such a relation is inconsistent and error could be reported.

CONDITIONAL CRITICAL REGIONS

- Region construct is not adequate for general synchronization. (useful only for critical section).
- Hoare then introduced the conditional critical region construct. [1972 Hoare, "Towards a Theory of Parallel Programming"]

Region V when B do S;

Here V is shared data

 B is a Boolean

 S is a Statement

- As before, regions referring to same shared variable (V in this definition) are to be mutually exclusive.
- However, here process enters a critical region and evaluates B and if true computes S.
- On the hand if B is found false, it relinquishes the mutual exclusion.
- It is quite interesting to see how conditional critical region could be implemented.

Producer-Consumer Problem using Conditional CR

Var buffer: shared record

begin

Pool: array[0..n-1] of item

Count, in, out: integer := 0;

end;

Producer inserts a new item nextp in buffer by executing:

Producer: Region buffer when count < n do

begin

pool[in] := nextp;

in := in + 1 mod n;

count := count + 1;

end;

Consumer: Region buffer when count > 0 do

begin

nextc := pool[out];

out := (out + 1) mod n;

count := count - 1;

end;

Producer-Consumer Problem using Conditional CR

Implementation of conditional critical region by semaphore is quite tricky and it involves various eventualities.

(Assume a FIFO ordering of waiting processes on semaphores).

- We need to evaluate B in critical section (mutual exclusion).
- If B is false then mutual exclusion is to be relinquished (given up).

The following data structures are used

Var

X_mutex : semaphore := 1;---mutual exclusion semaphore

X_wait : semaphore := 0;---wait semaphore

(if entry not possible due to B=false)

X_count : integer := 0;---No. of processes waiting on X_wait

X_temp : integer := 0;---No. of processes allowed to test their B

Producer-Consumer Problem using Conditional CR

Region X when B do S;---statement to compile using semaphores
(or a semaphore implementation of the program)

P(X_mutex) ---mutual exclusion

if not B then

begin

X_count := X_count + 1;---one more process to wait on P(X_wait)

V(X_mutex); ---relinquishes mutual

P(X_wait); ---exclusion and waits here. If allowed to go

further checks for B and if false does this.

While not B do

begin

X_temp := X_temp+1;---one more process allowed here

If X_temp < X_count ---all of us not in while loop

Producer-Consumer Problem using Conditional CR

i.e. some are

then $V(X_wait)$ at $P(X_wait)$ so signal them to
go ahead
else $V(X_mutex)$; -None of insiders can go ahead so take
out side.

$P(X_wait)$;

End;

$X_count := x_count - 1$;

End;

S;

If $X_count > 0$

then begin

$X_temp := 0$;

$V(X_wait)$; ---makes insider to proceed

end;

else $V(X_mutex)$;---outsider to come in

Producer-Consumer Problem using Conditional CR

- Hoare's construct allows delaying only at beginning of the Region.
- Hansen thus modified the construct as:

Region V do

begin

S1;

await (B);

S2;

end;

- When a process enters the critical region it computes S1 (S1 be null). If B is true, S2 is executed.
- On the other hand if B is false mutual exclusion is given up and the process is delayed until B becomes true.

CLASS CONCEPTS

- ❖ Programming language supporting the class concept

Type class_name = class

Variable declarations..

Procedure P1(...);

begin....end;

Procedure P2(...);

begin....end;

:

:

Procedure Pn(...);

begin....end;

begin ----class initialization,

initialization code

end;

CLASS CONCEPTS

- All procedures defined in a class need not be visible to the user of class.
- All variables declared need not be visible to the user of class.
- Good language must support these requirements.
- Procedure which are required to be visible outside may be written as:
- Procedure Entry P(...);
 - Procedure, without the keyword entry are not accessible from out side.
- Original class concept did not have the control over the visibility of data declared in class.

CLASS CONCEPTS

EXAMPLE:

Use of class concept and Hansen's Region statement for coding 2nd reader/writers is illustrated now.

Type Reader_writer =class

Var V: shared record

nreaders, nwriters: integer;

busy: Boolean;

end;

nreaders, nwriters: integer;

Procedure entry open_read;

begin

Region V do begin

await (nwriter = 0)

nreaders := nreaders + 1;

end;

end;

CLASS CONCEPTS

Procedure entry close_read;

begin

Region V do begin

nreaders := nreaders - 1;

end;

end;

Procedure open_write;

begin

Region V do begin

nwriters := nwriters + 1;

await ((not busy) and (nreaders = 0));

busy := true;

end;

end;

CLASS CONCEPTS

```
Procedure close_write;
    begin
        Region V do begin
            nwriters := nwriters - 1;
            busy := false;
        end;
    end;
begin
    busy := false;
    nreaders := 0;
    nwriters := 0;
end;
```

Usage

rw: Reader_writer;---object rw is created

---Note that entire environment is created with initialization

rw-> as an object created by the variable declaration above

outsiders refers to the name as:

rw.open_read;

rw.v etc.

rw.open_read; (Open a file)

:

:

read file

rw.close_read;

:

:

rw.open_write;

...

write

rw.close_write;

CLASS CONCEPTS

1. Every reader/writer process must observe the above sequence of operations.
2. Many readers/writers may execute the above code.
3. Our implementation with conditional critical region (Hansen's) takes care of how to and whom to grant access.
4. Note that every process wanting to operate on a file (example above) must use the above sequence. Any violation of above will lead to errors.

CLASS CONCEPTS

Class implementation above does not guarantee the misuses like

- ❖ A process might operate on a file without gaining access (by `open_read` or `open_write`).
- ❖ A process may not release the file it got access.
- ❖ A process may release a file it never opened.
- ❖ Many such programming errors in a process could cause problems for others.
- ❖ Note that these difficulties are similar to those that motivated us to develop the concept of critical region.
- ❖ Previously we have to worry about correct use of semaphores.
- ❖ Now we have to worry about correct use of higher level operations.

CLASS CONCEPTS

One way to ensure that readers/writers observe the appropriate sequences is to incorporate shared file itself within the class with only two entry procedures read and write. Open_read, open_write, close_read, close_write are local procedures and not entry procedures.

```
Type Reader_writer = class
    Var V: shared record
        nreaders, nwriters: integer;
        busy: Boolean;
end;
Procedure open_read;
begin
    Region V do begin
        await (nwriters = 0);
        nreaders := nreaders + 1;
        end;
    end;
Procedure close_read;
begin
    Region V do nreaders := nreaders - 1;
end;
```

CLASS CONCEPTS

```
Procedure open_write;  
  begin  
    Region V do begin  
      nwriters := nwriters + 1;  
      await ((not busy) and  
            (nreaders = 0));  
      busy := true;  
    end;  
  end;
```

```
Procedure close_write;  
  begin  
    Region V do begin  
      nwriters := nwriters - 1;  
      busy := false;  
    end;  
  end;
```

```
Procedure Entry read;  
begin  
    open_read;  
    ...  
    read a file  
    close_read;  
end;
```

```
Procedure Entry write;  
begin  
    open_write;  
    ...  
    writing..  
    ...  
    close_write;  
end;
```

CLASS CONCEPTS

- ❖ Note now that, read, write are the only procedures users see.
- ❖ Class implementation takes care of the protocol.
- ❖ No outsider (to class) use open procedures.
- ❖ Protocol is automatically maintained by the code of class.
- **Drawback** – The class procedures read/write not to the user requirement. The code is fixed by class writers, who may not allow possible uses of users.
- ❖ For example, user may want to do reading 5 times in one shot.
- ❖ We have seen critical region mechanism for process synchronization.
- ❖ By combining this with class concept we have obtained a greater degree of control over the object usage.
- ❖ One unattractive feature here is that every process has to define its own synchronization

CLASS CONCEPTS

This lead to the definition of an object called as monitor as a language construct

- ❖ It allows safe and effective sharing of abstract data types
- ❖ Syntax is similar to the class concept with additional semantics that monitor assures mutual exclusion.
- ❖ Monitors are similar in many respect to the critical region and thus are not adequate for general
- ❖ Additional mechanism provided are condition variables. It is an abstract data type with two operations, wait, and signal

❖ X:condition

❖ X.wait → a process executing this is suspended

❖ X.signal → awakens a process sleeping on X (if none sleeping, like NOP)

❖ X.signal contrast to V operation where state of semaphore is always affected

- P executes X.signal , and Q is suspended
- P waits until Q leaves monitor or waits for another signal
- Q waits until either P leaves or waits for another signal

Both conditions are equally good

CLASS CONCEPTS

Implementation of binary semaphore using monitor

Type semaphore=monitor

Var busy:Boolean

non busy:condition

Procedure entry P;

Begin if busy then non busy.wait End;

Procedure entry V

Begin busy:=false;

non busy.signal;

End;

Begin

busy:=false;

End;

CLASS CONCEPTS

➤ Dining Philosopher Solution using Monitor

Basis: Philosopher is allowed to pick-up chopstick if both are available.

➤ Data structure:

Var state:array[0..4] of (thinking,hungry,eating);

Self:array[0..4] of condition

Philosopher I can set state[I]=eating only if its neighbors are not eating, and can delay himself when he is hungry.

i.e. state[I-1 mod 5] <> eating and state[I+1 mod 5] <>eating

Type D_P=monitor

Var state:array[0..4] of (thinking,hungry,eating);

self:array[0..4] of condition

Procedure entry pickup(I:0..4)

Begin

State[I] := hungry; test(I);

If state[I] <> eating then self[I].wait;

End;

CLASS CONCEPTS

Procedure Entry putdown(I:0..4);

Begin

State[I]:=thinking;

Test(I-1 mod 5); test(I+1 mod 5);

End;

Procedure test(K:0..4);

Begin

If state[K+4 mod 5] \neq eating &

State[[K+1 mod 5] \neq eating then

Begin

State[k]:=eating;

Self[k].signal; End;

Begin for I:=0 to 4 do state[I]:=thinking]; End;

Var dp:D_P;

Philosopher executes

dp.pickup(I)

Eat; Dp.putdown(I);