Concurrent Programming

S1: a = b + c;
S2: d = e * f;
S3: g = a | d;
S4: h = g * i;

Read set = { b, c, e, f, a, d, g, i }

Write set = { a, d, g, h }

→ S_1 & S_2 complete its oper^n independently.

→ But S_3 depends on result of S_1 & S_2.

→ & S_4 also executed only after completion of S_3.

Dependency Graph
(or)
Precedence Graph

→ S_1 & S_2 execute concurrently bcz they dont have any common variable.

Read set ∩ Write set = ϕ

→ Any two statements S_i and S_j can be executed concurrently or parallely if and only if the below cond^{ns} are followed -

1) $R(S_i) \cap W(S_j) = \phi$
2) $W(S_i) \cap R(S_j) = \phi$
3) $W(S_i) \cap W(S_j) = \phi$

} donot have any common variable

⇒ The concurrent program will be written by using the below statements -

parallel — par begin — par end

(or)

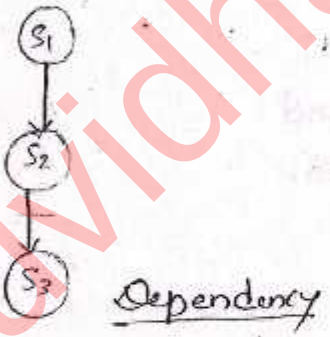
co begin — co-end

i

concurrent

Ex:-

```
begin
    S1;
    S2;
    S3;
end
```



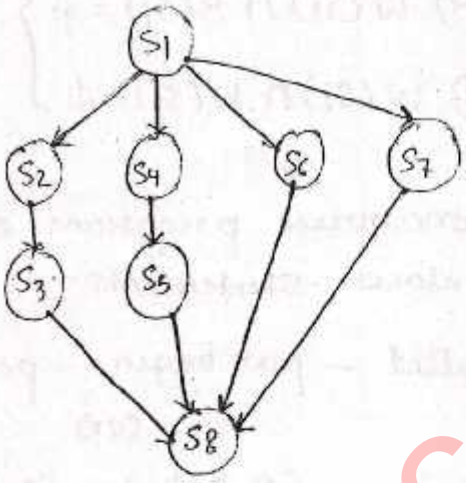
Dependency graph

```
par begin
    S1;
    S2;
    S3;
par end
```

Dependency graph

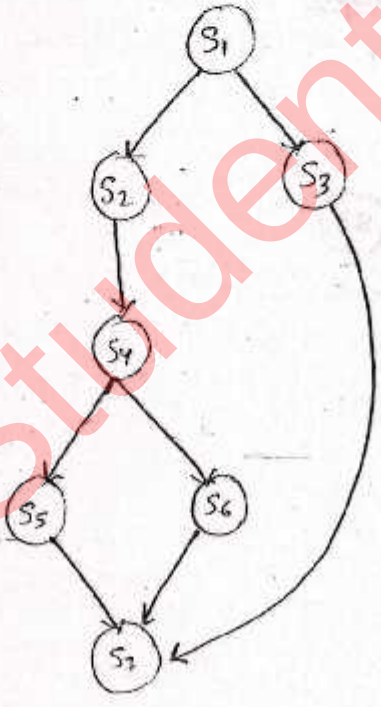
Q? begin — if nothing is given here, by default it is begin & End

```
S1;
par begin
    begin
        S2;
        S3;
    end
    begin
        S4;
        S5;
    end
    S6;
    S7;
par end
    S8;
end
```



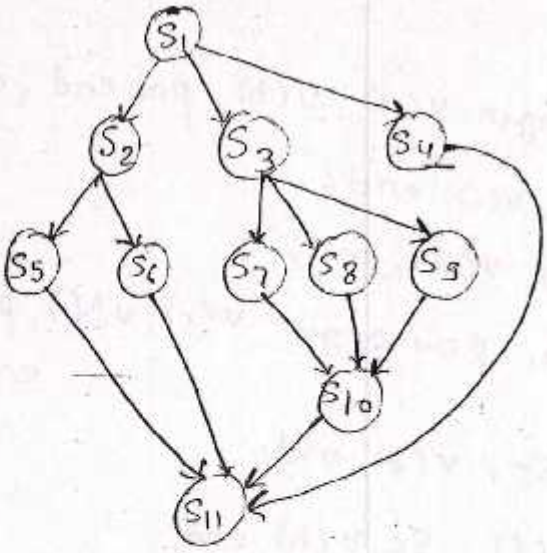
Dependency graph

Q.1




```
S1;
par begin
    begin
        S2;
        S4;
        par begin
            S5;
            S6;
        par end
    end
    S3;
par end
    S7;
```

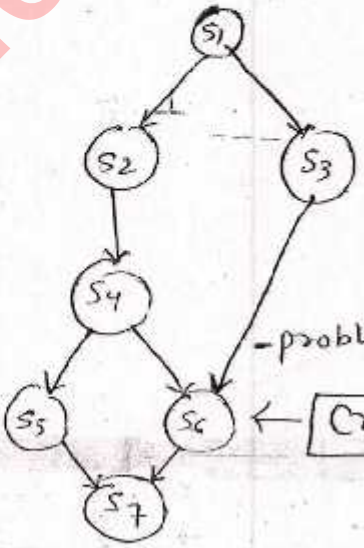
Q.2




```
S1;
parbegin
  begin
    S2;
    parbegin
      S5;
      S6;
    parend
  end
  begin S3;
    parbegin
      S7;
      S8;
      S9;
    parend
    S10;
  end
  S4;
parend
S11;
```

⇒ It is not possible to write concurrent programming for all the dependency graph using par begin and par end. But it is very much possible with the help of semaphore operations

Qu?



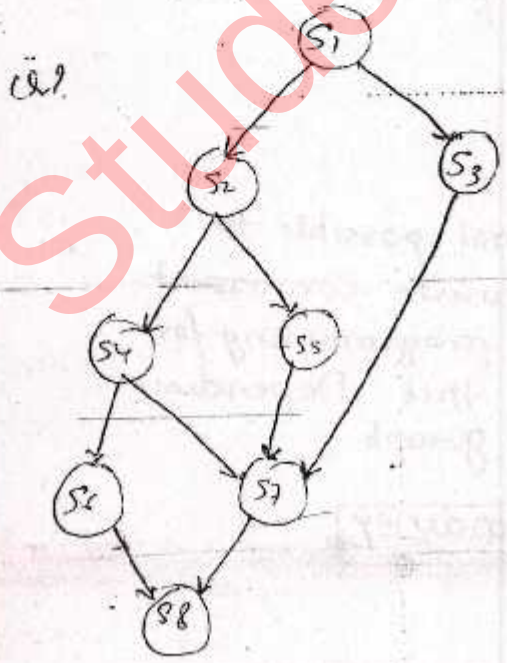
not possible to write concurrent programming for this Dependency graph.

```
par begin
    begin S1, parbegin V(a), V(b), parend, end;
    begin p(a), S2, V(c), end;
    begin p(b), S3, V(d), end;
    begin p(c), S4, parbegin V(e), V(f), parend,
                                              end;
    begin p(e), S5; V(g), end;
    begin p(d), p(f), S6, V(h), end;
    begin p(g), p(h), S7, end;
par end
```

All the binary semaphore variables a, b, c, d, e, f, g, h are initialized to '0'.

p(a) - is successful only when S_1 is completed.

→ If we alter the statements, no problem arises, when the whole (entire) statements will be executed as bcz the entire dependencies are maintained by semaphore oper^n.



```
par begin
    begin S1, par begin V(a), V(b), par end, end;
    begin P(a) S2, par begin V(c), V(d), par end, end;
    begin P(b), S3, V(e), end,
    begin P(c), S4, par begin V(f), V(g), par end, end,
    begin P(d), S5, V(h), end,
      begin P(f), S6, V(i), end,
    begin P(e), P(g), P(h), S7, V(j), end,
    begin P(i), P(j), S8, end,
par end
```

Grate One

```
int x=0, y=0;
par begin
    begin
        x=1;
        y=y+x;
    end
    begin
        y=2;
        x=x+3;
    end
par end
```

What can be the final values of 'x' and 'y' after completing the concurrent program.

I. x=1, y=2

II. x=1, y=3 ✓

III. x=4, y=6 ✓

Which of the above claims are possible?

(a) only I & II

(b) only II & III

(c) only I & III

(d) all I, II & III

Sol:- $x=0, y=0$

→ $x=1$

$y=y+x=0+1=1$

$y=2$

$x=x+3=1+3=4$

$\boxed{\begin{matrix}x=4\\y=2\end{matrix}}$

→ $x=1$ ← pre-empt

$y=2$

$x=x+3=1+3=4$

$y=y+x=2+4=6$

$\boxed{\begin{matrix}x=4\\y=6\end{matrix}}$

→ $x=1$

$y=2$

$y=y+x=1+2=3$

$x=x+3=1+3=4$

→ $y=2$

$x=x+3=0+3=3$

$x=1$

$y=y+2=2+1=3$

$\boxed{\begin{matrix}x=1\\y=3\end{matrix}}$

→ $y=2$ ← pre-empt

$x=1$

$y=y+x=2+1=3$

$x=x+3=1+3=4$

$\boxed{\begin{matrix}x=4\\y=3\end{matrix}}$

→ $y=2$

$x=1$

$x=x+3=4$

$y=y+x=6$

fork() system call Implementation

```
main()
{
  int pid;
  pid = fork();
  if (pid<0)
  {
    printf ("fork failed");
  }
  else
  {
    if (pid==0)
    {
      printf ("child process");
    }
    else
    {
      printf ("parent process");
    }
  }
}
```

⇒ The fork() is a system calls which is used to create the child process.

⇒ The fork() returns a -ve value, if the child process creation is unsuccessful.

⇒ The fork() returns the value '0' to the newly created child process.

⇒ The fork() returns the +ve value {process id of the child process} to the parent process.

⇒ fork() will return an integer type of variable.

Ex

```
main()
{
  fork();
  printf("Hello");
}
```

parent process

```
main()
{
  printf("Hello");
}
```

child process

(Exactly copy of parent process except fork())

```
main
{
  fork();
  fork();
  printf("Hello");
}
```

```
main
{
  fork();
  printf("Hello");
}
```

child

```
main
{
  printf("Hello");
}
```

child

```
main
{
  printf("Hello");
}
```

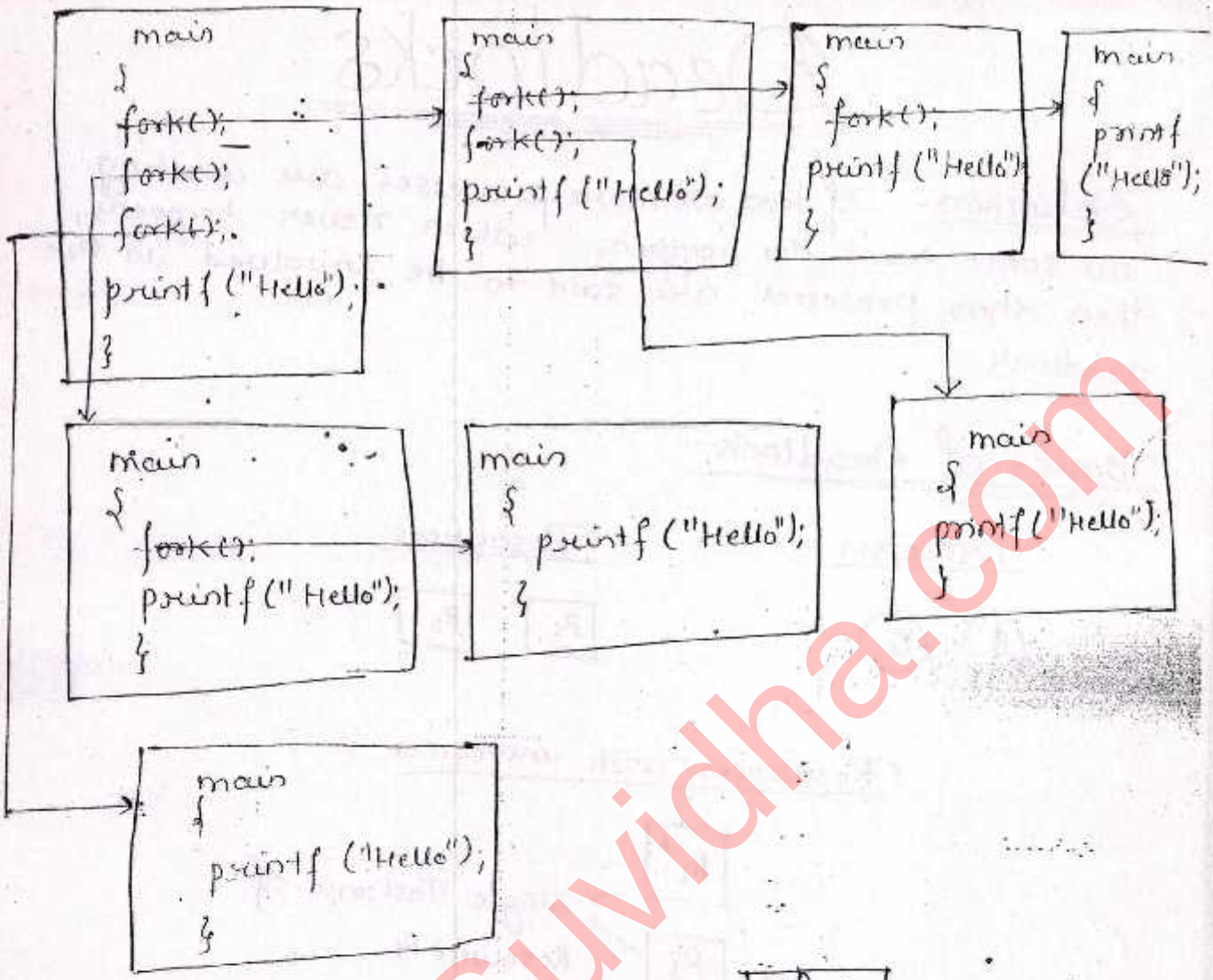
child

4 times - hello will print

3 child process

no. of child process = $2^n - 1$

no. of process = 2^n



no. of child process = 7 $2^n - 1$

no. of times printf execute = 8 2^n

(where n is the no. of fork calls.

⇒ When the child processes is created by using the fork() system call, the parent & child processes will have the below addresses—

(i) Relative address — will be same for both. p & c.
(ii) Absolute address — will be diff° for both.

⇒ Programmatically if we try to print the addresses - it will print relative address by using symbol '&'.