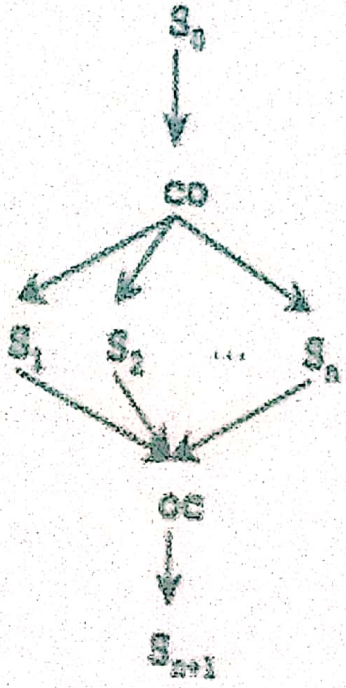
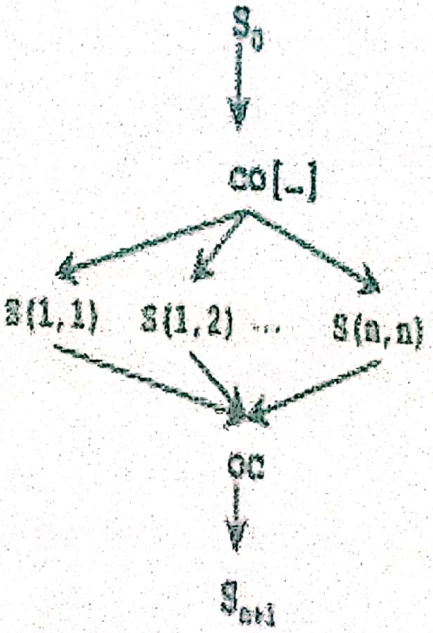
Two records of the co




```
S0;
co S1; # thread 1
|| ...
|| Sn; # thread n
oc;
Sn+1;
```




```
S0;
co (i=1 to n, j=1 to n) { # n x n threads
    S(i,j);
}
oc;
Sn+1;
```

Example of non-deterministic behavior of a parallel program

- Program:

```
int y = 0; z = 0;
co x = y + z; ||  y = 1; z = 2; oc
```

- Possibility 1:

```
x = y{0} + z{0}; y = 1; z = 2;      {x == 0}
```

- Possibility 2:

```
y = 1; x = y{1} + z{0}; z = 2;      {x == 1}
```

- Possibility 3:

```
y := 1; z := 2; x := y{1} + z{2};   {x == 3}
```

- Possibility 4:

```
load y{0} to R1; y := 1; z := 2;
  add z{2} to R1{0}; store R1 to x; {x == 2}
```

Example <await (B) S;>

- Parallel program:

```
int x = 1, y = 2, z = 3;
co x = x + 1;
|| y = y + 2;
|| z = x + y;
|| <await (x > 1) x = 0; y = 0; z = 0;>
oc
```

- Does the parallel program end?
- What are the final values of variables x, y and z?
- Let's not forget: the addition is not an atomic operation, but we must first read the value from memory into the register, add it, and then write it back to the memory.

  - Program ends.
  - Possible values for variables are:

```
x == {0}
y == {0,2,4}
z == {0,1,2,3,4,5,6}
```

<await (s>0) s = s-1;>

The command waits until s is not positive, then s is reduced by 1 (s decrease only when it is positive. Nothing happens in between)

<await (s>0);>

The command waits until s is not positive.

<s=s-1;>

Atomic command

Example of using the <await (B) S;> command and if

Look at this example:

```
int x = 0;
co <await (x != 0) x = x - 2 >;
|| <await (x != 0) x = x - 3 >;
|| <await (x == 0) x = x + 5 >;
oc
```

Is the parallel program coming to an end, if yes what are possible values for variable x, if not, why? Change <await> with if (delete <>). Now program is definitely over. What are possible values of variable x now?

- (a) Program finishes. End value of **x = 0**. Last process executes first, then the first two in arbitrary order.

```
int x = 0;
co if (x != 0) x = x - 2;    //S1
|| if (x != 0) x = x - 3;    //S2
|| if (x == 0) x = x + 5;    //S3
oc
```

conditions S1 and S2 are false, S3	`x == 5`
condition S2 is false; S3, S1;	`x == 3`
condition S1 is false; S3, S2;	`x == 2`
S3; S1 || S2	`x == {0, 2, 3}`

Semaphores

P - Try; V - Release (from Dutch: Proberen and Verhogen)

A Semaphore is a data structure that includes one integer variable on which two atomic operation are defined:

- "Try to enter" P(s)
- "Release" V(s)

```
P(s):   < await (s > 0) s = s - 1; >
V(s):   < s = s + 1; >
```

$s \geq 0$

Declaration and Initialization:

- sem s; — semaphore with value s = 0
- sem s = expr. — semaphore with value s=expr.
- sem s[1:n] ([n] expr) — table of semaphores s[n] = expr.

Implementation:

Implementation of P(s)

```
if (s > 0) s = s-1;
else {
      stop the process and set it in
      waiting queue (sem_wait_queue)
      }
```

Implementation of V(s)

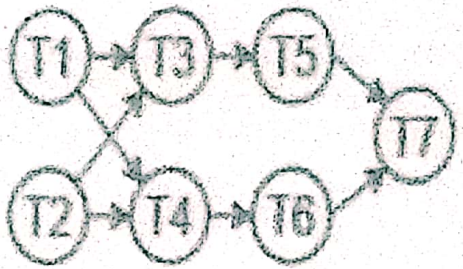
```
if ( empty(sem_wait_queue) ) s = s+1;
else {

      Take the first process from sem_wait_queue
      and puts it in line for execution
      ready_queue
      }
```

Semaphores

Exercise:

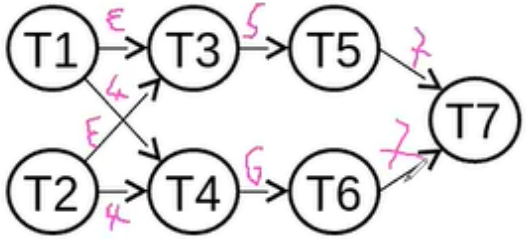
The correct order of execution should be implemented/ensured using semaphores. Suppose that in order to solve a problem, it is necessary to perform tasks in the order shown in the given graph



Suppose that every task is performed through a process:
It means, for example: T5 waits on execution of process T3 and after execution (of process T3) it proceed to process T7.

```
process Ti (i = 1, ..., 7) {
    wait for predecessors, if any;
    execute the task;
    signal successor, if any;
}
```

Write a program that will perform tasks with processes in the order shown on the graph. Synchronize between processes with semaphores. Minimize the number of semaphores.



semaphor[3] will be used to communicate between T1 and T3.
Also same s[3] will be used communicate between T2 and T3.
V(S[3]) => Raising the flag for T3. When both V(S[3]'s raised we can continue with T3.
it can be directly executed, doesn't have to wait for anything

```
sem S[3:7] = ([5] 0);                    // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3]); ... V(S[5]);
T4:: P(S[4]); P(S[4]); ... V(S[6]);
T5:: P(S[5]); ... V(S[7]);
T6:: P(S[6]); ... V(S[7]);
T7:: P(S[7]); P(S[7]); ...
```

it must be waken up 2 times because there are 2 P(S[7]).

T5 sleeps on 5, then signals 7

Since T7 is executed over T3 and T4, then we can use one of those two semaphores for T7, example: S[3]

we use one less data structure but it's not good.

```
sem S[3:6] = ([4] 0);                    // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3]); ... V(S[5]); V(S[5]);
T4:: P(S[4]); P(S[4]); ... V(S[6]);
T5:: P(S[5]); ... V(S[3]);
T6:: P(S[6]); ... V(S[3]);
T7:: P(S[5]); P(S[3]); P(S[3]); ...
```

Exercise:

Suppose that code is given, where X is integer value.

```
int a = X; sem s1=0;
<a = a+1;>
<a = a+2;>
<a = a * (a+3);>
```

Programmer was trying to create parallel code and wrote the code in next format:

```
int a = X; sem s1=0;
co <a = a+1;>
|| <a = a+2;>
|| <a = a * (a+3);>
oc
```

Unfortunately result of the parallel code was not same as result of the sequential code. Set the minimum number of operations P and V (over already defined semaphore s1) so that program always return correct result.

Minimum number of P+V is 4, for example:

```
int a = 1; sem s1 = 0;
co <a = a + 1;> V(s1);
|| <a = a + 2;> V(s1);
|| P(s1); P(s1); <a = a * (a + 3);>
oc
```

Exercise:

The command a = b + c is implemented with machine commands and is mapped to atomic commands temp = b + c; a = temp; where *temp* is a local variable of the process(each process has its own local variable). Write all possible traces and final value for each of them.

```
int x = 1; int y = 2;
co x = y + x;
|| y = x + 1;
oc
```

The code with atomic commands is:

```
int x = 1; int y = 1;
co TEMP=y+x; x=TEMP;
|| TEMP=x+1; y=TEMP;
oc
```

And possible traces and solutions are:

1	1	2	2	x=2	y=3
1	2	1	2	x=2	y=2
1	2	2	1	x=2	y=2
2	1	2	1	x=2	y=2
2	1	1	2	x=2	y=2
2	2	1	1	x=3	y=2

Exercise:

```
int x = 0, y = 10, z = 6
co x = x + z;
|| y = x - y;
|| y = 9;
oc
```

Write all possible solutions

TH1			TH2			TH3	
1	R1	x	4	R2	x		
2	operation	R1=R1+z	5	operation	R2=R2-y	7	y=9
3	store	x=R1	6	store	y=R2		

1	2	3	4	5	6	7
R1=0	R1=0+6	x=6	R2=6	R2=-4	y=-4	y=9

x	y	z
6	9	6

This is one solution of given program

If there are n threads such that $Thread_i$ executes m_i atomic actions, then the number of possible interleavings (options/solutions) of the atomic actions is:

$$\frac{(m_1+m_2+\cdots+m_n)!}{m_1!\cdot m_2!\cdots m_n!}$$

Some more explanations:

At-Most-Once condition is a condition under which expression evaluations and assignments will appear to be atomic. A critical reference in an expression is a reference to a variable that is changed by another thread.

At-Most-Once:

- An assignment statement x=e satisfies the At-Most-Once property if either:
  - e contains at most one critical reference and x is neither read nor written by another thread, or
  - e contains no critical references, in which case x may be read or written by other threads.
- An expression that is not in an assignment satisfies At-Most-Once if it contains no more than one critical reference.

```
int x=0; y=0
co x = y+1; //thread1
|| y=1;     //thread2
oc
```

- The expression in Thread1's (x=y+1) assignment statement references y (one critical reference) but x is not referenced by Thread2.
- The expression in Thread2's assignment statement has no critical references.

Both statements satisfy At-Most-Once.

if references other variable like y; then contains 1 critical reference.