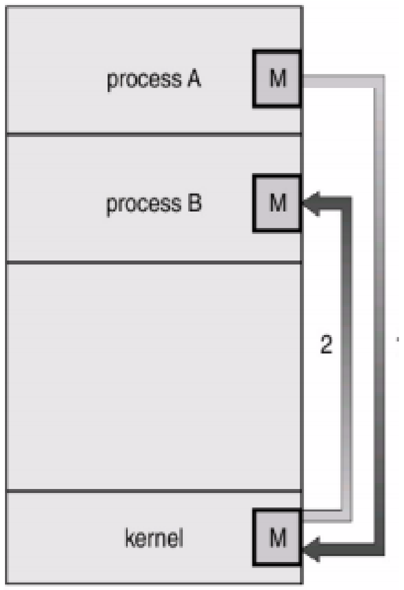
IPC WITH MESSAGE QUEUE

Processes can communicate by sending and receiving messages to each other. So, this is a very different kind of mechanism where various system calls or library functions which might be called during message send and receive. Therefore one process can package a piece of data within a message and send it the other process. As a result the other process can implicitly receive that message.

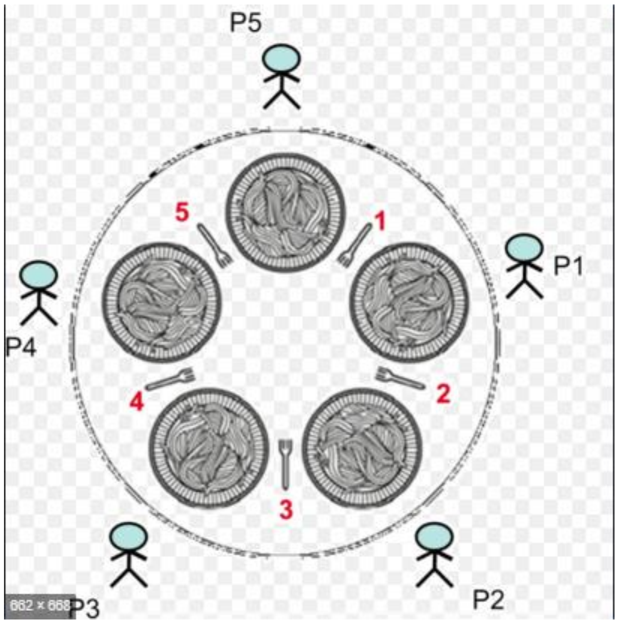


Important points:

1. Shared memory created in the kernel space (unlike the IPC with shared memory concept where it was created in the user space).
2. System calls are required such as SEND and RECEIVE used for communication between the two processes.
3. Advantage: explicit sharing and less error prone as kernel ménages the sharing.
4. Limitation: As we know that systems calls have their own individual overheads. So this mechanism is slow compared to shared memory.

PROCESS SYNCHRONISATION

Previously it is discussed that how semaphore works and how it helps in synchronizing processes. Now another famous classic problem is there which I am going to discuss: Dinning Philosopher's problem. A beautiful use of Semaphore is also seen here. So let us start with the problem.



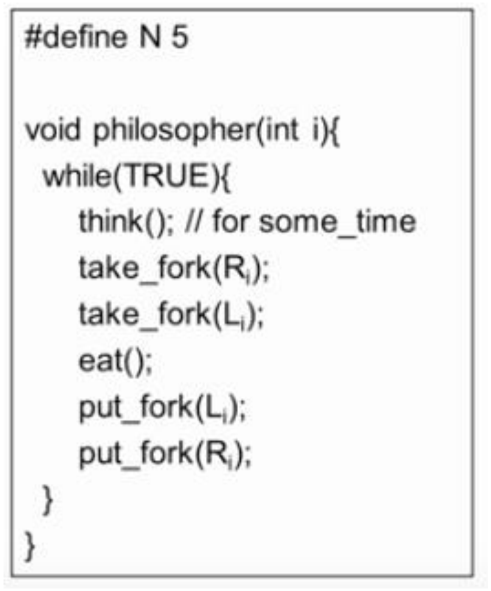
Here, P1,P2,P3,P4,P5 are the five philosophers who are sitting around a table. In front of every philosopher there is one plate and total five forks.

Now read the below mentioned conditions carefully:

1. Each of the philosophers can either think or eat (in other words, if the philosopher is not eating, he is thinking).
2. In order to eat, a philosopher needs to hold both forks (one on his left, one on his right).
   : Say for instance, if P1 wishes to eat then he need to hold fork 1 and fork 2 both, if P4 wishes to eat then he needs to hold fork 4 and fork 5 both.

PROBLEM STATEMENT: Develop an algorithm where no philosopher starves.
i.e : every philosopher can get a chance to eat .

FIRST ATTEMPT TO BUILD ALGORITHM:



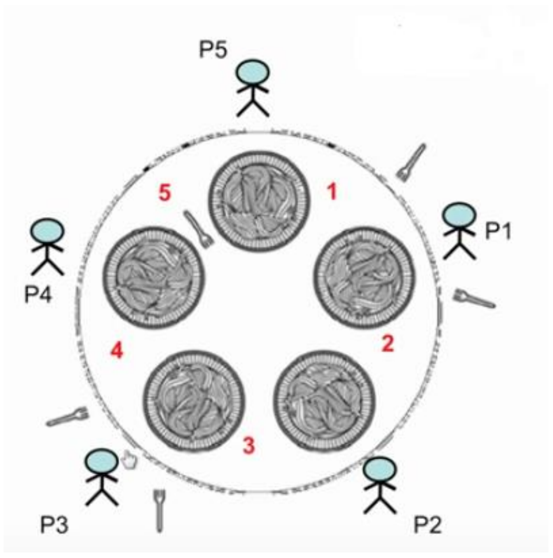
```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     take_fork(R_i);
     take_fork(L_i);
     eat();
     put_fork(L_i);
     put_fork(R_i);
  }
}
```

Here, we have defined N as 5 corresponding to each philosopher. The function for philosophers takes an integer I which can be 1 to 5 corresponding to each philosopher (P1,P2,P3,P4,P5). Now an infinite loop is there where the ith philosopher will think for some time then after some time he begins to feel hungry, so he will take his right fork then he will take his left fork and will eat for some time. After this after finishing he will put down his left fork then he will put down his right fork.
THIS CONTINUES IN AN INFINITE LOOP.

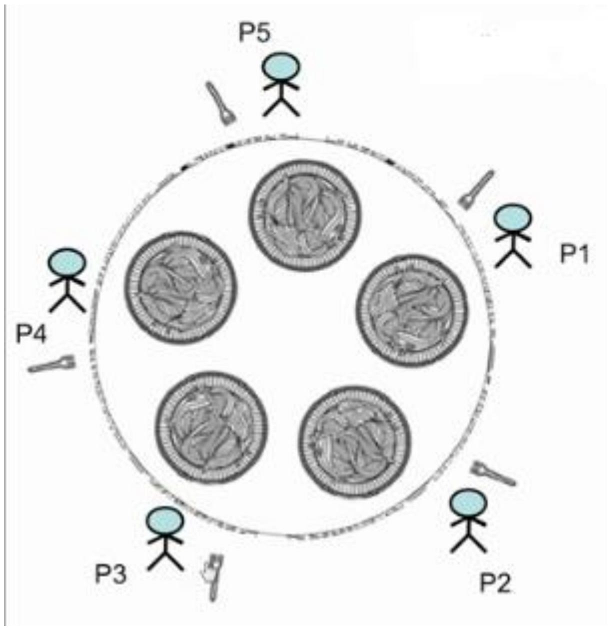
It seems really easy as a solution to this problem but there are some certain issue which can pop up.

- ISSUES: SCENARIO 1



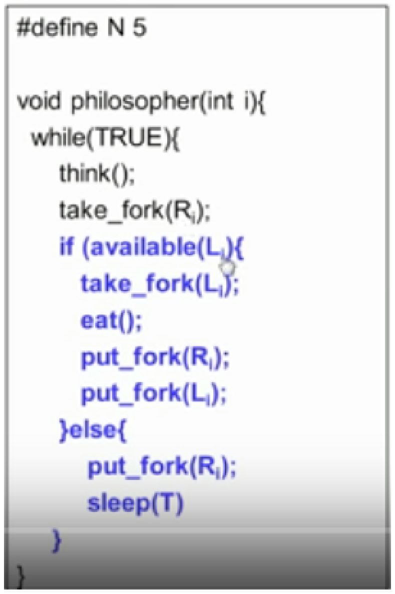
In this scenario, P1, P3 have a higher priority,i.e when they wish to eat system ensures that they get the forks. While the other philosophers P2, P4, P5 have the lower priority, will not be able to eat. So, P2 have neither the right fork nor the left fork and therefore P2 will not able to eat. In the similar manner P4 & P5 is having one fork between them which they could possibly pick up. The other fork In each case, with high probability will be given to the philosophers P1 & P3.**Thus P2, P4, P5 will starve, which is not at all a desirable situation that should take place.**

➢ <u>ISSUES: SCENARIO 2</u>



All the philosophers pick up their right fork simultaneously. Now in order to eat each of the philosophers have to pick up the left fork, and this could lead to a starvation.
For example for philosopher P1 is waiting for P2 to put down the fork it could pick it up, thenP2 is waiting for P3 ,then P3 is waiting for P4 to put down the fork , then P4 is waiting for P5 and P5 is waiting for P1.So essentially every philosopher is waiting for the other philosopher to perform eating, and thus they are creating a waiting chain here. This waiting can go till infinite, leading to starvation. **This particular scenario is called deadlock**.

<u>SECOND ATTEMPT TO BUILD ALGORITHM:</u>



```
#define N 5

void philosopher(int i){
  while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
    }else{
      put_fork(Ri);
      sleep(T)
    }
}
```

We have the same function like the previous attempt. This time the philosopher takes the right fork. Then he would determine that the left fork is available or not. If available, the left fork is taken and eat for some time, then the both forks are put down.in the else part if the left fork is not available the philosopher will also put down the right fork so that the other philosophers can try to eat. After this is done the particular philosopher will go for sleep for some fixed time ‘T’ before trying to eat again.

Let us now consider the issues regarding this solution:

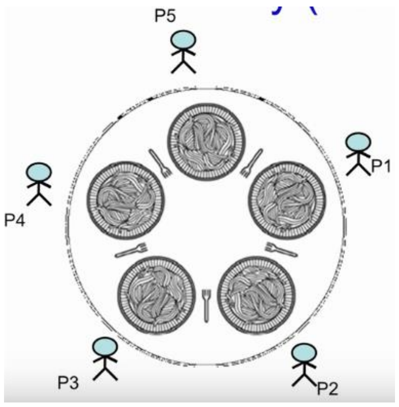
- <u>ISSUES: SCENARIO 1</u>

Imagine all the philosophers

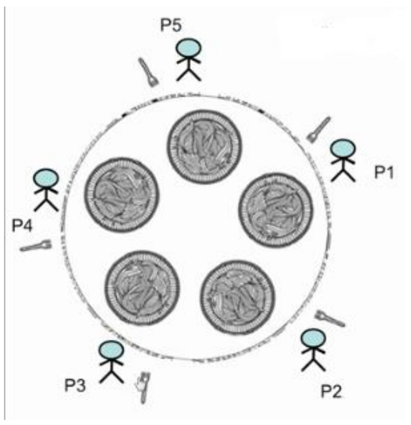
- **Start at the same time**
- **Run simultaneously**
- **And think for the same time**

This could lead a visual like below: where basically no philosopher will finally get a chance to eat. It shows the same scenario of heading towards a deadlock condition.

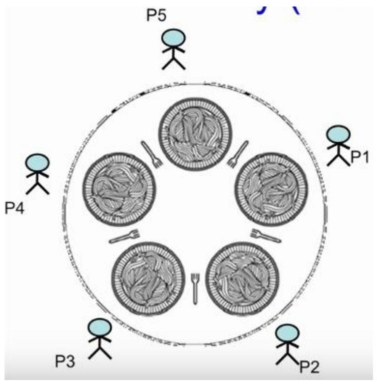
START AT THE SAME TIME



RUN SIMULTANEOUSLY



AND THINK FOR THE SAME TIME



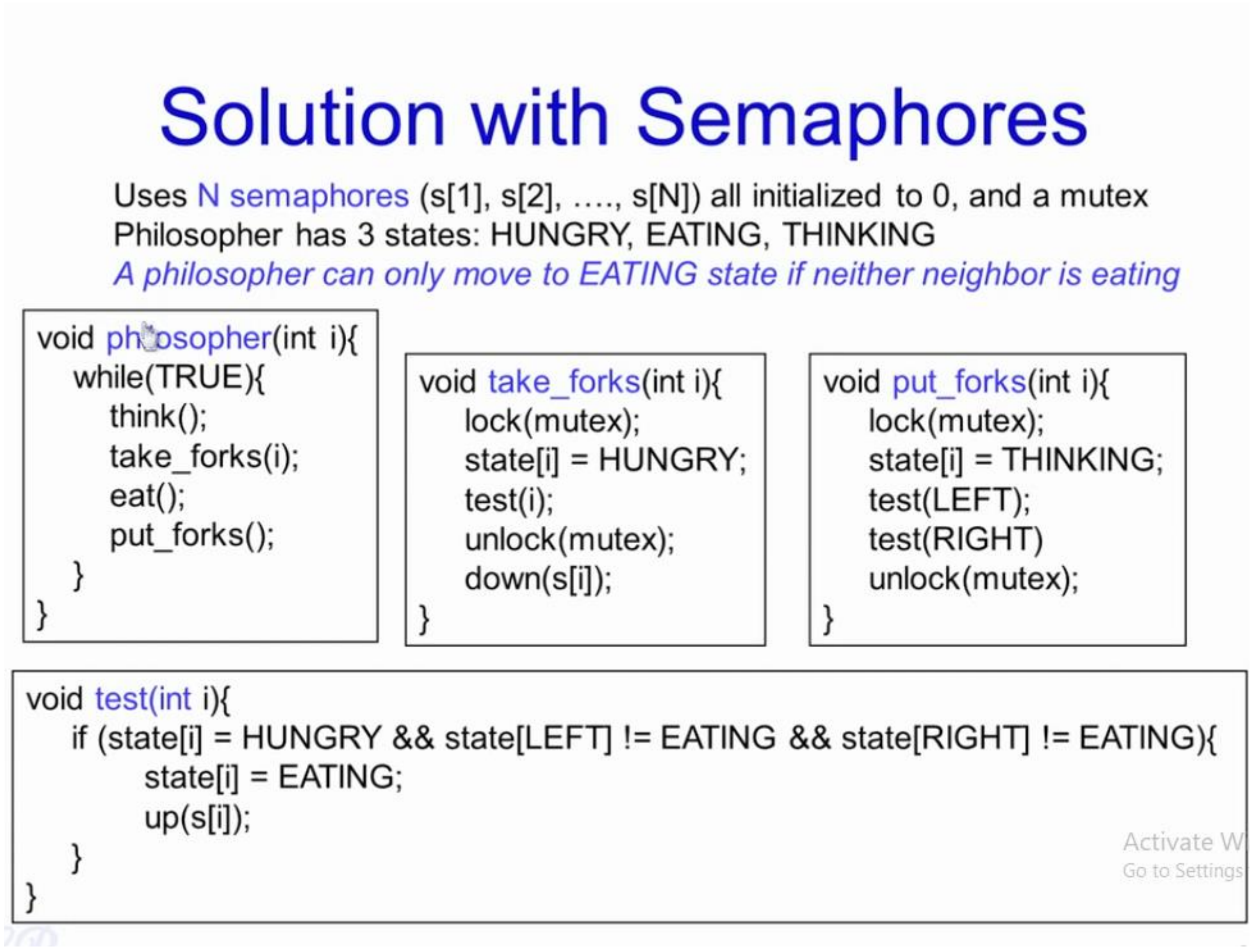
After penetrating into the second try vividly and understanding the issues of this solution a better solution can be drawn from this second attempt that if the sleep time quantity type is changed from fixed to random then the chances of philosophers facing starvation will be less, though this is not a permanent solution.

In the next step we are going to discuss a new attempt to solve this problem with using locks like semaphore and mutex.

<u>THIRD ATTEMPT TO BUILD ALGORITHM:</u>

So in this attempt of solution to the dinning philosophers problem more than one semaphore is taken as you can see. There is one semaphore each for every philosopher sitting around the table .

The philosophers are suggested to have three states through which they can move. Like they can be in HUNGRY state for some time, they can switch to EATING state for some time and then lastly it can undergo THINKING state for some time.



Solution with Semaphores

Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
   while(TRUE){
      think();
      take_forks(i);
      eat();
      put_forks();
   }
}
```

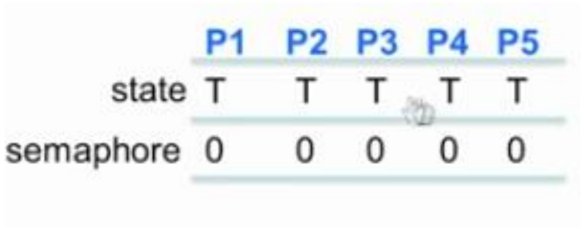
```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
          state[i] = EATING;
          up(s[i]);
    }
}
```

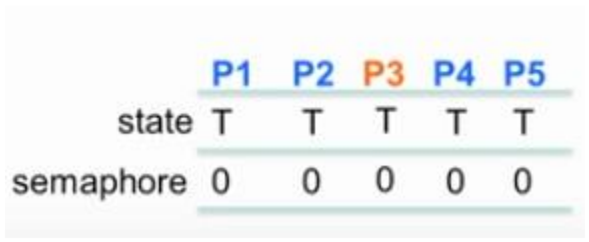
EXAMPLE :

Step 1: All the philosophers are at THINKING state.

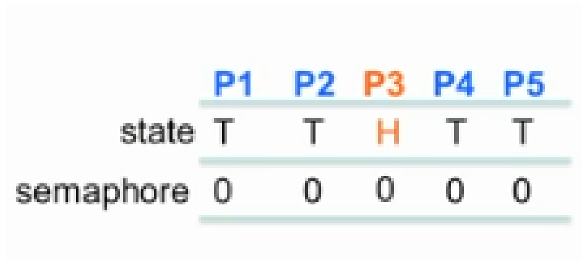


	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

Step 2: Suppose P3 philosopher wants to eat , so after entering into *void philosopher(int i)* function it enter into *take_forks(i)* Function . You can see inside this function the state of P3 becomes HUNGRY by executing the statement "state[i] = HUNGRY".

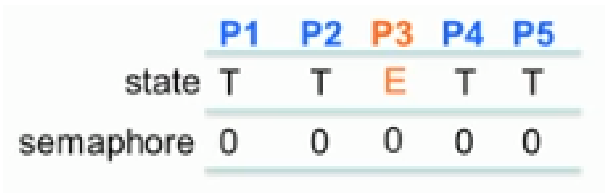


	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0



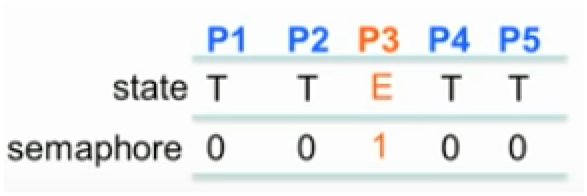
	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

Step 3: Now test(i) is called . So P3 enters inside the void test(int i) function. Here the if condition returns true which signifies the left philosopher of P3(which is P2) & the right ohiolosopher of P3(which is P4) both are not eating , so as a result P3 goes into EATING state,

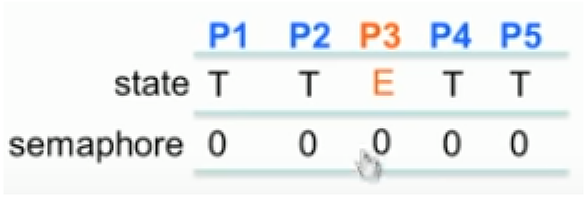


	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

Step 4: s[i] is a semaphore which will be set to 1 at UP(s[i]) and will be decremented by 1 and set to 0 again by executing down(s[i]).



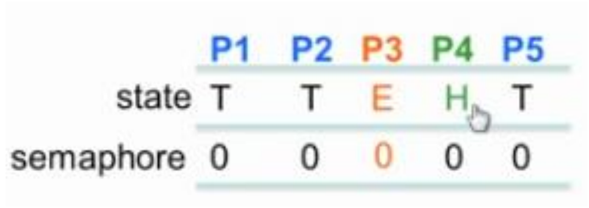
	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0



	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

So after this the philosopher will execute eat () operation and will finish eating.

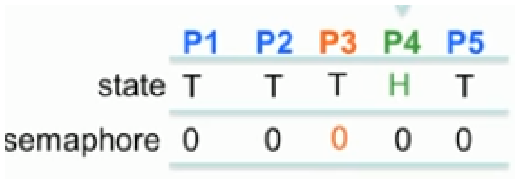
Step 5: suppose when P3 is in eating state at that time if P4 enters inside the void philosopher(int i) and switches the state from THINKING to HUNGRY .when P4 will execute test(int i) , if condition will return false. Then coming out of test function it will try to complete the operation down(s[i]) but it won't be possible as P4 could not perform UP(s[i]) prior to this. So philosopher will get block at down function call of semaphore.



	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

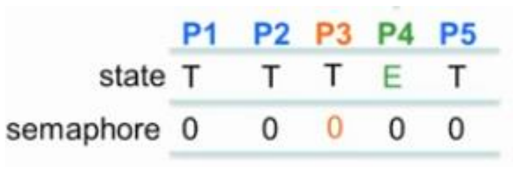
Step6 : Now when P3 will complete eating state and will execute put_forks()function then P3 will switch the state from EATING to Thinking . Now it will invoke test (LEFT) where result wont effect the situation as LEFT philosopher of P3 is P2 which is in THINING state. But in next line when Test(RIGHT) is invoked, as right philosopher is P4 then now It will execute void test (int i), where if condition will return true and will get a chance to execute UP, switch the state from HUNGRY to EATING and so on.

1.



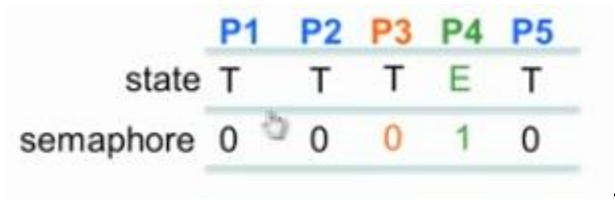
	P1	P2	P3	P4	P5
state	T	T	T	H	T
semaphore	0	0	0	0	0

2.



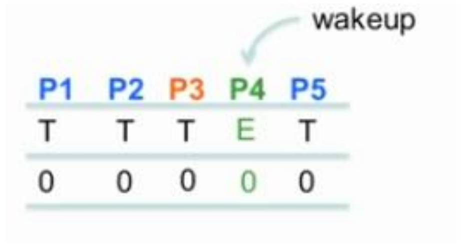
	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	0	0

2.



	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

4.



P1	P2	P3	P4	P5
T	T	T	E	T
0	0	0	0	0