**Fork-join Paranoia:**

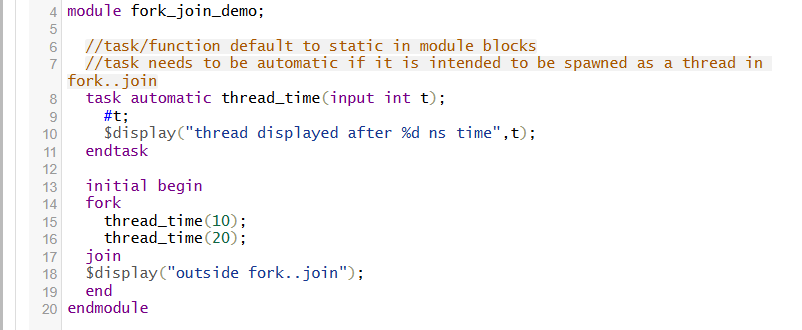
**We will look at mainly three aspects here:**

1)fork….join, fork….join\_any, fork….join\_none variants

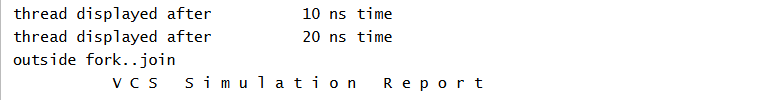
2)Process control – wait fork, disable, disable statement

3)Inter-process synchronization & communication (events, semaphores, mailboxes)

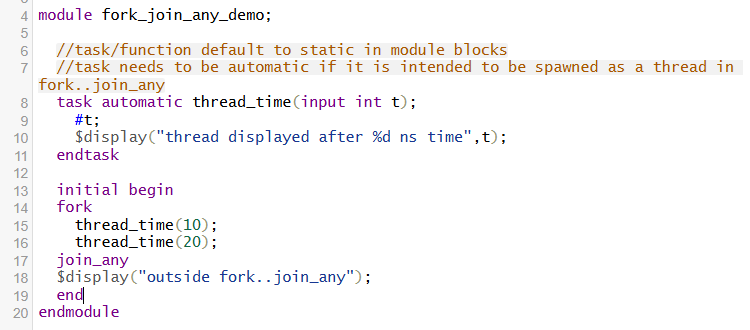
**1)fork….join:**



**Output:**



**2)fork…join\_any**

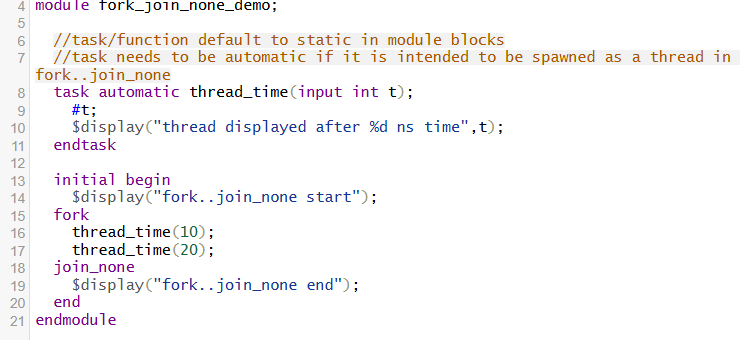


**Output:**

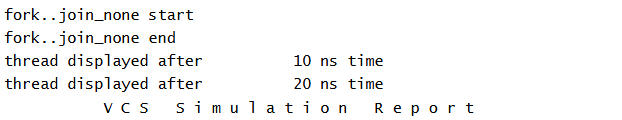


**3)fork..join\_none:**

🡪This construct enables running threads in background. The child threads are spawned/scheduled by fork..join\_none but they don’t start executing until the ***parent thread blocks (using a #0, wait fork etc.)*** **OR** ***parent thread terminates (following fork..join\_none block)***



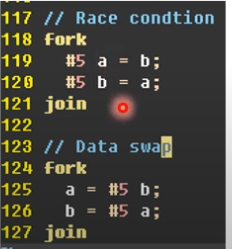
**Output:**



**\*\*Intra-assigned delay operator behavior in fork..join:**

**A = #5 b;** //Intra-assigned delay (b is evaluated at time=0 & assignment of A happens after t=5ns0

🡪*Behaviour similar to non-blocking assignment (RHS evaluated in active region & assignment of RHS to LHS postponed until NBA region)*

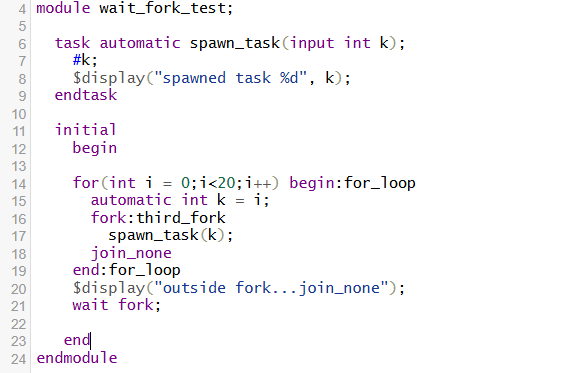


**2)Process control:- wait fork, disable, disable fork**

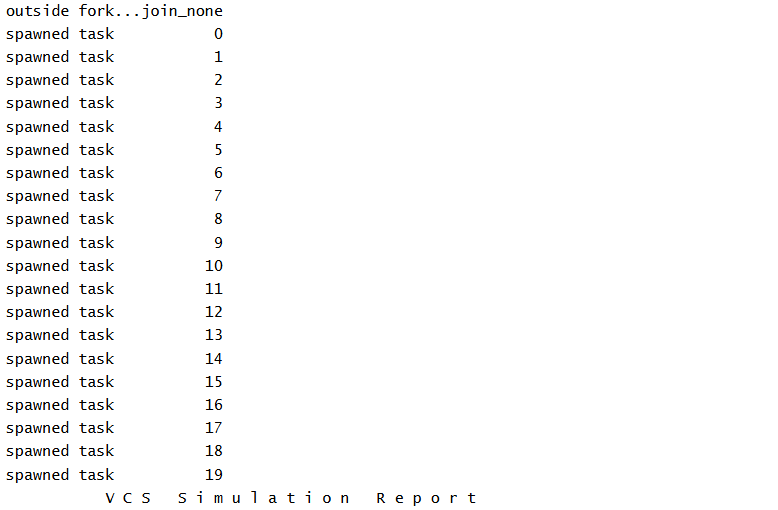
**🡪wait fork (real-life execution):**

The wait fork statement blocks process execution flow ***until all immediate child subprocesses*** (processes created by the current process, excluding their descendants) have terminated.

E.g: Creation of 20 threads using fork..join\_none inside for loop:



**Output:**



**🡪disable/ disable fork:**

**1)**disable fork needs to be used outside fork..join construct as it is meant to destroy/terminate the all processes (immediate child+descendents) under fork..join

2)disable label can be used inside fork..join,begin..end as it is tied to the label. With disable label one can destroy/terminate itself.

More examples tomorrow….

a)how to disable a specific thread in fork..join? Labels can be used inside fork..join to kill a specific thread.

**3)IPC & Synchronization:**

**a)Semaphore:**

🡪Conceptually Semaphore is a bucket.

🡪When a semaphore is allocated, a bucket that contains a fixed number of keys is created.

🡪Processes using semaphores shall first procure the key from the bucket before they can continue to execute.

🡪If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously.

🡪Semaphores are used for mutual exclusion(one thread at a time), access to shared resources & basic synchronization.

🡪Shared resources between threads.

Important info:

1)number of keys created == number of concurrent threads/processes allowed to access it. (total number of keys created using new())

2)default ***get()*** is single key but>1 can also be specified

🡪if keys are not available the calling process gets blocked (until keys are available again)

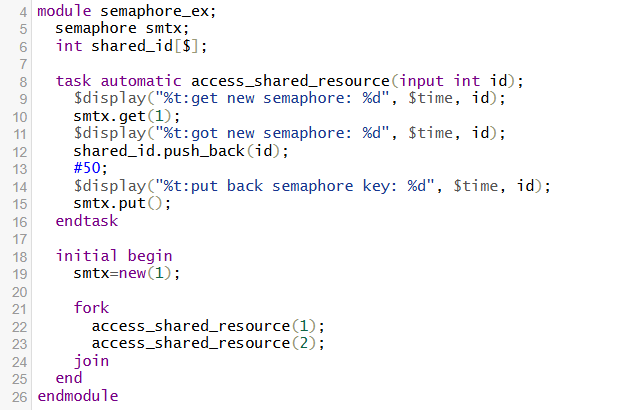
🡪Waiting queue behaves as FIFO.

3) default ***put()*** is for single key but>1 can also be specified

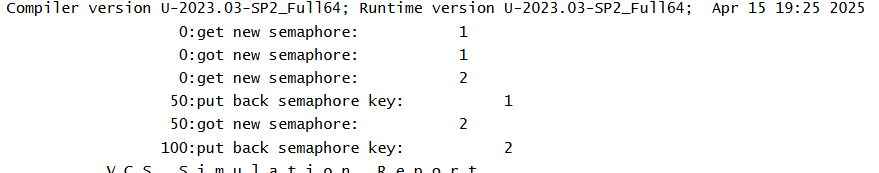
🡪put call returns the keys

🡪suspended process resumes if sufficient number of keys are returned

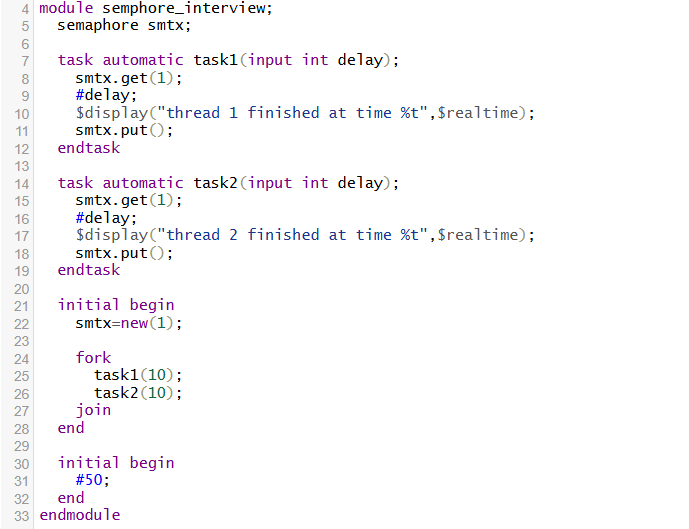
**Basic Ex-1:-** (Credits:- Anand Shirahatti)



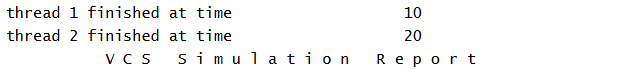
Output:



**Interview question-2:**



**Output:**



**2)Named Events:**

Race condition definition: - one block is reading something at the same time some other block is writing to it. These both are happening at the same time. So do we get the old value or new value?

How can a thread block, and wait for another thread to wake it up?

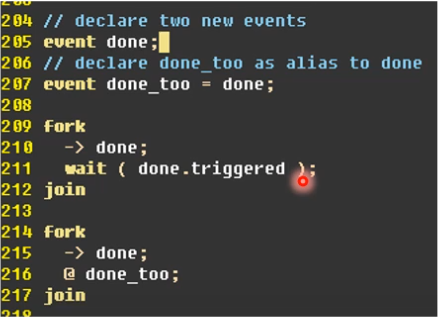
🡪The event data type is a momentary flag (a simulation event)

🡪Synchronize activity between concurrent procedural blocks

🡪(🡪) triggers an event in the active region – blocking event trigger

🡪(**->>**) triggers event in the NBA region – non-blocking event trigger

Ex-1: **(Credits Anand Shirahatti)**



The second fork (starting from line 214-217) has a implicit bug.

Since the order in which the processes execute is unknown in fork..join, hence as far as the process in line 216(@-waiting for event “done”) if executes first then it remains blocked until the event is triggered in line 215 after which line 216 can execute smoothly (as far as this is happening in current time-step).

However, if the process in line 215(->trigger event done) executes first then by the time the process in line 216 executes(@waiting of event done to trigger), the trigger might get lost & hence the code will be blocked forever.

In code snippet from line 214-217 – Even when we were executing processes in line 215 & 216 in the same time step, but since the ordering of the processes under fork..join was indeterministic we weren’t able to capture the event.

To resolve the issue mentioned above:

Let’s look at a workaround where we wait for the triggered state when waiting for an event to trigger.

Look at code snippet from line (209-212):

Irrespective of the order in which the two processes in line 210 & 211 get executed & even if wait statement gets executed after the event done is triggered, because we are in the same time step(1ns) the done.triggered state will remain true until we advance into next time step.(2ns)

***However, if somehow(theoretically) the code snippet from line 209-212 executes with zero delay multiple times then the triggered state will not get reset but will remain stuck at 1 unless we advance to the next time step – practically not ideal***