

In class we examined the need for **concurrent execution paths** like a **consumer** and a **producer** to synchronize their access to a **shared ring buffer**.

Below are a set of **global objects** which are accessible to a **single producer** thread and a **single consumer** thread.

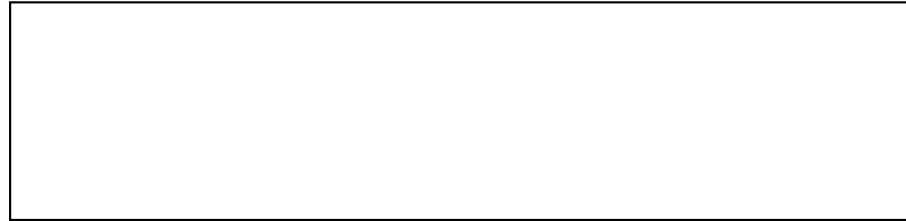
You must write a solution using **semaphores** with the semaphore declaration format shown. This format requires you to fill in the initial semaphore values in your declarations. You must declare and initialize however many semaphores you need to solve this problem efficiently. The shared ring buffer is an array of **10 integer locations**.

The producer must execute a **forever loop** using a random number function (like **random()**) to create an integer and then place the integer into an appropriate slot in the shared ring buffer **when it's safe to do so**.

The consumer must execute a **forever loop** taking numbers out of the shared ring buffer and printing them to standard out (with a **printf()** type function) **when it's safe to do so**. Using **C code style**, write the **producer function** and the **consumer function as described above**, given the simple semaphore functions **p()** and **v()** whose prototype headers are declared below the global data. **Busy-waiting** is not allowed anywhere in your solution.

GLOBALS TO PRODUCER AND CONSUMER THREADS:

```
semaphore_type sem_name = sem_initial_value; ← format  
DECLARE YOUR SEMAPHORE(S) HERE
```



```
int ring_buffer[10], in = 0, out = 0;  
void p ( semaphore_type * );  
void v ( semaphore_type * );
```

WRITE PRODUCER FUNCTION HERE

WRITE CONSUMER FUNCTION HERE



- . Use a **collection of CSP processes** to recursively compute the **nth** power of the number **x** (both **n** and **x** are considered to non-negative integers).

The program should consist of:

```
[F(0):: F(1)!x; F(1)!n; F(1)?result t;
```

```
||
```

```
  F(i : 1.. (maxexp+1)):: *[ ..WRITE THIS CODE BELOW.. ]  
]
```



In class we discussed a synchronization example called the **observer - reporter problem**. An **observer process** can see something as it passes by a sensor and wants to increment a **shared global counter** for each passing object. A reporter process spends most of its time sleeping, but every so often it **awakens**, sends **the current object count** found in the shared counter to a printer, and then **resets the shared counter to 0**. While either the reporter or the observer is using the counter the other process must be kept away to avoid corrupting the counter.

- You must code this problem in 'C' style for **both the observer and reporter** as **void functions** called observer and reporter as shown:

```
void observer ( void );
```

```
void reporter ( void );
```

using the fewest number (if any) of **eventcounters** and **sequencers** possible, but using **no busy-waiting**. The reporter should use the standard 'C' library routine **int sleep (int seconds);** to delay his reporting for **15 minutes** between reports. The following types and operations are available:

```
ec_t event_counter; // declare EC, value defined 0  
seq_t sequencer; // declare SEQ, value defined 0  
void await ( ec_t * , int ); // await event  
void advance (ec_t * ); // advance EC  
int ticket (seq_t *); // get a SEQ ticket
```

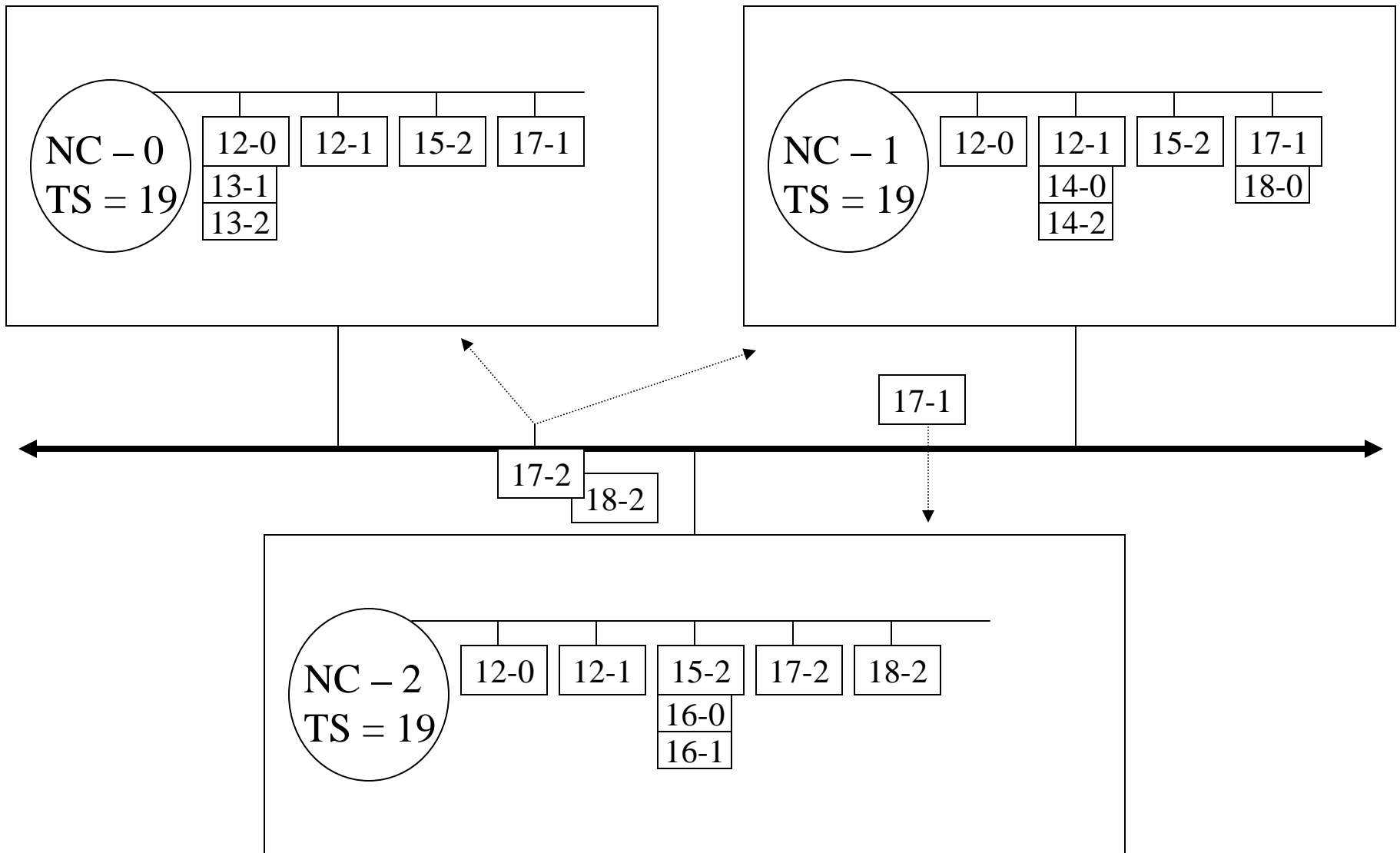
Show the declaration of the event **counter(s) and sequencer(s)** (if any) you need and any global variables that will be shared by both the observer and the reporter as global declarations (with initialization where needed), and then code each of the observer and reporter functions.

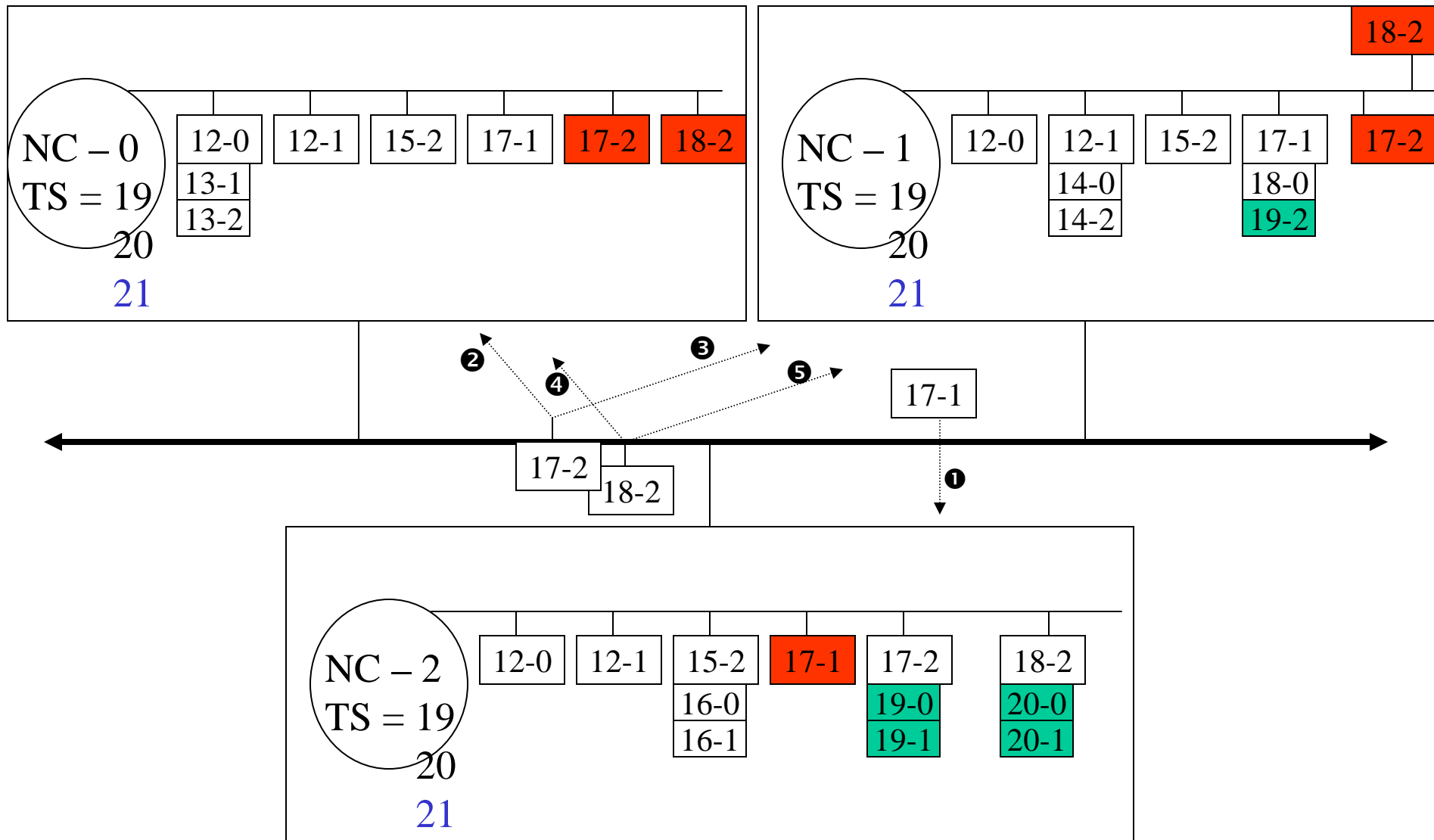
GLOBALS TO OBSERVER AND REPORTER:

WRITE OBSERVER FUNCTION HERE

WRITE REPORTER FUNCTION HERE







Consider the following resource-allocation policy for a fixed inventory of **serially reusable** resources of three different types (such as tape drives, printers, shared memory, etc.):

- **Requests and releases** of resources are allowed **at any time**.
- If a request for a resource is made by a process which is **already holding other resources**, the request may be **denied** based on a system imposed **ordering** required for allocations. For example, in a system imposed ordering it may be required that any process that must hold a tape drive and a printer at the same time must ask for and obtain the tape drive(s) before asking for the print device(s).
- Resources which are **currently in use by other processes** will cause a requesting process to block waiting for their availability in FIFO order.
- Whenever a resource is **freed**, some blocked process needing such resource may secure the resource and **move to the ready state**.

- A.** List the **4 necessary conditions** for a deadlock to occur in a computing system.
- B.** Can **deadlock** occur in the system described above ? **If so**, give an example. **If not**, which **necessary condition** cannot occur that would be required for a deadlock ?
- C.** Can **indefinite postponement** occur ? **Explain.**

The following information depicts a system consisting of 3 processes (**a, b, and c**) and **10 tape drives** which the processes must share. The system is currently in a "**safe**" state with respect to deadlock:

process	max tape demand	current allocation	outstanding claim
a	4	2	2
b	6	3	3
c	8	2	6

Following is a sequence of events, each of which occurs a short time after the previous event with the first event occurring at time one (t(1)). The exact time that each event occurs is not important except that each is later than the last. I have marked the times **t(1), t(2)**, etc. for reference. Each event either **requests or releases** some tape drives for one of the processes. If a system must be kept "**safe**" at all times, and if a request can only be met by providing all the requested drives, indicate the time at which each request will be granted using a **first-come-first-served** method for any processes that may have to wait for their request (i.e. request 5 granted at t(9)) or indicate that a request will not be granted any time in the sequential times listed. (Note: if a process releases some drives at time(x) which a waiting process needs, that waiting process will get its drives **at that time(x)**. Put your final answers in the space provided below.

<u>time</u>	<u>action</u>
t(1)	request #1 c requests 2 drives
t(2)	request #2 a requests 2 drives
t(3)	release a releases 3 drives

ANSWERS:

Request #1 occurs at _____

Request #2 occurs at _____