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:

```c
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
GLOBAL TO PRODUCER AND CONSUMER THREADS:
sem_t sem_name = sem_initial_value; ← format

DECLARE YOUR SEMAPHORES HERE
sem_t prod = 10;
sem_t cons = 0;

int buf[10], in=0, out=0;
void p ( sem_t * );
void v ( sem_t * );

WRITE PRODUCER FUNCTION HERE                       WRITE CONSUMER FUNCTION HERE

void producer(){
while(1){
    p(&prod);
    buf[in] = random();
    in = (in + 1) % 10;
    v(&cons);
}
}

void consumer(){
    int val;
    while(1){
        p(&cons);
        val = buf[out];
        // print val somewhere
        out = (out + 1) % 10;
        v(&prod);
    }
}
Weak Reader Preference Solution to the Reader-Writer Problem Using Even-counters and Sequencers

ec_t rdE, wrE; // event counters always init 0
seq_t rdS, wrS // sequencers always init 0
int nreaders = 0; // count of current readers

void reader_in(){
    await(&rdE, ticket(&rdS));
    if(nreaders == 0)
        await(&wrE, ticket(&wrS));
    ++ nreaders;
    advance(&rdE);
} // reader can read

void reader_out(){
    await(&rdE, ticket(&rdS));
    --nreaders;
    if(nreaders == 0)
        advance(&wrE);
    advance(&rdE);
} // writer can write

void writer(){
    await(&wrE, ticket(&wrS));
    // writer can write
    advance(&wrE);
}
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:

  ```c
  void  observer ( void );
  void  reporter ( void );
  ```

using the fewest number (if any) of **mutex locks** and **condition variables** from the pthread library, 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:

```c
pthread_mutex_t   a_mutex;              // declare mutex
pthread_cond_t      a_condx              // declare a condition variable
int pthread_mutex_lock(pthread_mutex_t *)        // lock mutex
int pthread_mutex_unlock(pthread_mutex_t *)    // unlock mutes
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *)   // wait on condx
int pthread_cond_signal(pthread_cond_t *)         // signal condx
```
Show the declaration of the mutex and condition variables (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. You don’t have to initialize any possible mutex or condition variables, assume they are already initialized when you use them.

**GLOBALS TO OBSERVER AND REPORTER:**

WRITE OBSERVER FUNCTION HERE

WRITE REPORTER FUNCTION HERE
The “Observer – Reporter” problem: an observer thread and a reporter thread must keep their counter use coherent using a single pthread mutex

GLOBALS TO OBSERVER AND REPORTER:

```
pthread_mutex_t    obj_lock;
int                obj_count = 0;
int   pthread_mutex_lock(pthread_mutex_t   *mlock);
int   pthread_mutex_unlock(pthread_mutex_t   *mlock);
```

OBSERVER THREAD FUNCTION
```
void *observer (void * arg){
  while (1){
    // when object passes
    pthread_mutex_lock(&obj_lock);
    ++obj_count ;  // increment counter
    pthread_mutex_unlock(&obj_lock);
  }  // while
}  // observer
```

REPORTER THREAD FUNCTION
```
void *reporter (void * arg){
  while (1){
    sleep (900);  // sleep for 15 min
    pthread_mutex_lock(&obj_lock);
    obj_count = 0;       // reset counter
    pthread_mutex_unlock(&obj_lock);
  }   // while
}   // reporter
```
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:

<table>
<thead>
<tr>
<th>process</th>
<th>max tape demand</th>
<th>current allocation</th>
<th>outstanding claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(1)</td>
<td>request #1</td>
<td>c requests 2 drives</td>
</tr>
<tr>
<td>t(2)</td>
<td>request #2</td>
<td>a requests 2 drives</td>
</tr>
<tr>
<td>t(3)</td>
<td>release</td>
<td>a releases 3 drives</td>
</tr>
</tbody>
</table>

**ANSWERS:**
Request #1 occurs at _______

Request #2 occurs at _______
<table>
<thead>
<tr>
<th>process</th>
<th>tape demand</th>
<th>current allocation</th>
<th>outstanding claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>action</th>
<th>details</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(1)</td>
<td>request #1</td>
<td>a requests 1 drive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t(2)</td>
<td>request #2</td>
<td>c requests 2 drives</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t(3)</td>
<td>release</td>
<td>b releases 1 drive</td>
</tr>
</tbody>
</table>

4 2 2  t1
6 3 3
8 2 6
3  R1 OK t1

-----------------------------
4 3 1  t2
6 3 3
8 2 6
2  R2 WAIT t2

-----------------------------
4 3 1  t3
6 2 4
8 2 6
3  R2 OK t3

-----------------------------
4 3 1  t4
6 2 4
8 4 4
1
T4 and T5 in DL