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 **<u>single</u> producer** thread and a **<u>single</u> 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**

DECALRE 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

A single producer, single consumer ring buffer, synchronized with **counting** semaphores

GLOBAL TO PRODUCER AND CONSUMER THREADS:

```
sem_t  prod = 10;  // semaphore initialized  to 10 spaces
sem_t  cons = 0;    // semaphore initialized  to 0 objects

int buf[10], in=0, out=0;  // 10 element ring buffer and pointers
void p ( sem_t  * );          // available p() and v() functions
void v ( sem_t  * );
```

PRODUCER FUNCTION

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

CONSUMER FUNCTION

```
void consumer(){
int val;
while(1){
   p(&cons);
   val = buf[out];
   // print val somewhere
   out = (out + 1) % 10;
   v(&prod);
}
```

Contemporary operating systems like **Windows and Linux/UNIX** may provide several **scheduling policies** to meet various thread scheduling needs, but often the default scheduling policy for non-privileged threads is a time-sharing (**TS**) policy known as **HPF/RR**.

A. What does **HPF/RR** stand for ?

Highest Priority First / Round Robin

B. Threads that are scheduled with **real-time policies** like the **POSIX FIFO** policy are generally **treated differently** than time-sharing (**TS**) threads in **two ways**. **First,** their priorities are generally **always higher** than any **TS** thread (they start off at a higher number than the highest possible **TS** thread). **What is the second major difference** in the way the system treats such threads ?

The Operating System does NOT do Dynamic Priority Adjustment (Aging)

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

GLOBALS TO OBSERVER AND REPORTER:

```
ec_t        my_ec;
seq_t       my_seq;
int         obj_counter = 0;
```

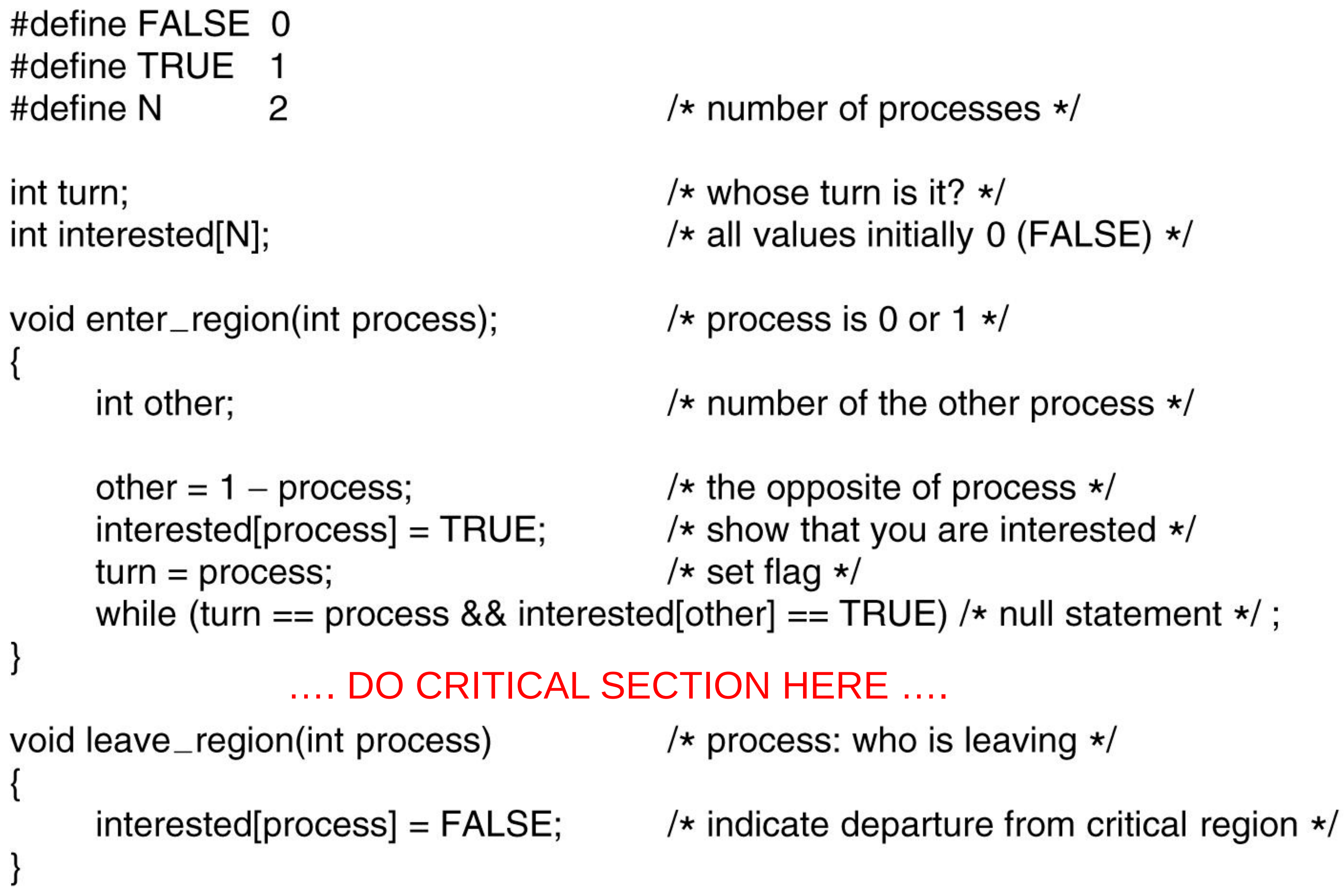
WRITE OBSERVER FUNCTION HERE

```
void observer (void){
  while (1){
    // when object passes
    await (&my_ec, tcket (&my_seq));
    obj_counter ++;
    advance (&my_ec);
  }   // while
}     // observer
```

WRITE REPORTER FUNCTION HERE

```
void reported (void){
  while (1){
    sleep (900);
    await (&my_ec, tcket (&my_seq));
     // Print obj_counter somewhere
    obj_counter = 0;
    advance (&my_ec);
  }   // while
}     // reporter
```

Be prepared to detect some change to Peterson’s Algorithm and how such a change would affect its behavior



```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}
```

…. DO CRITICAL SECTION HERE ….

```
void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

The following complete program shows a parent process creating **a single pipe and child.**

As you can see, the child is programmed to run a **grep "123"** command after redirecting its standard input to come from the pipe. The parent will enter a loop and **create 200 random integers, convert each integer into a string with a newline at the end of each number, and write them down the pipe, one line at a time**. The child grep will read its standard input one line at a time, looking for any lines that have the characters **"123"** in them, and print such a line to its **standard output**. (Assume all necessary include files are available; line numbers are for your reference)

In the following example, even though there are NO programming errors **and** NO system call errors, **the** parent **process** never completes. Explain **why the parent never finishes**, **and show** <u>where</u> **(using line numbers) and** <u>what code</u> **is necessary** for the parent to complete.

```c
int main(void){
    int pchan[2], pid, rval, cvar;
    char buf[20];
    if(pipe(pchan) == -1){
      perror("pipe");
      exit(1);
    }
    switch( pid = fork() ){
      case -1: perror("fork");
              exit(2);
      case  0: close(0);
                  if( dup(pchan[0] ) != 0 ){
                      perror("dup");
                      exit(3);
                  }
                  execlp( "grep", “grep”, "123", NULL );
                  perror("exec");
                  exit(4);
      default: for(cvar = 0; cvar < 200; cvar++){
                      rval = rand();                       // get random int
                      sprintf(buf, “%d\n”, rval);          // make string
                      write(pchan[1], buf, strlen(buf)); // write pipe
                  }
                  if(close(pchan[1])== -1 || close(pchan[0])== -1){
                        perror("close");
                        exit(4);
                  }
                  wait(NULL);
                  return(0);
    }  // switch
 }
```

Child does not close pchan[1] before execlp, thus grep will wait forever for EOF

The command **crypt** can be used on many UNIX systems to **convert clear text into encrypted text and to convert encrypted text back into clear text** using a common key**.** The beginning of the crypt man page is given below:

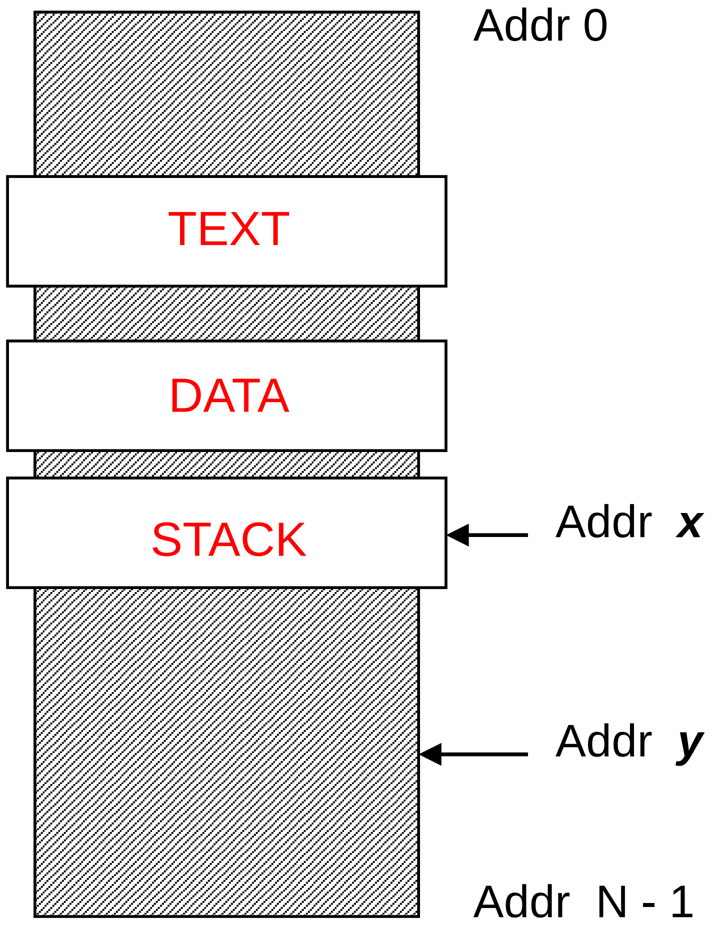
crypt(1)
NAME
  crypt - encode/decode
SYNOPSIS
  crypt key < input.File > output.File
DESCRIPTION
The crypt command reads from the standard input and writes on the standard output. You must supply a key which selects a particular transformation. The crypt command encrypts and decrypts with the same key.

If we assume that the parent process in the last slide now **wants the child to use the crypt command** to decrypt an encrypted file called **/tmp/mycypher** using the key **<u>keystring256</u>** and send the clear text up the pipe (in the last slide the parent wrote the pipe for grep to read, but here we want the child to write the pipe so the parent can read), then **show the necessary child code to do this** (i.e. provide the code we would need beginning at line 11 from the last slide). Show only the code you would need in this switch statement for case 0: (the child's code).

```
case  0: close(1);
 if( dup(pchan[1] ) != 1 ){
          perror("dup");
          exit(3);
 }
 if(close(pchan[0])==-1 || close(pchan[1])==-1){
          perror(“close”);
          exit(4);
 }
 close(0);
 if(open(“/tmp/mycypher”, O_RDONLY, 0) != 0){
          perror(“open”);
          exit(5);
 }
 execlp( “crypt”, “crypt”, “keystring256”, NULL );
          perror("exec");
          exit(4);
```

The diagram below depicts **a process virtual address space of N locations**, with 3 memory objects included within the address space. If the process was a part of the WindowsXP or UNIX operating system environments, these three objects would comprise a minimum requirement for an address space.

A. Label and briefly describe each of these required memory objects.



A. If the above address space represents a UNIX process, **describe** how a memory reference (such as a load instruction) to the address labeled Addr ***x*** would differ in behavior from a reference to Addr ***y***

x is a valid location, so reference succeeds
y is an invalid location, so reference causes a segmentation fault

B. If the above address space represents a UNIX process, **how will this address space change** if the process adds **2 more threads** ??

2 new stacks would be mapped into the address space

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.

Mutex Resources
Hold and Wait
No Pre-emption
Circular Wait

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 ?

NO DL, the Circular Wait Condition is Denied by imposing strict allocation ordering

C. Can **indefinite postponement** occur ? **Explain.**

NO IP … IP is a problem when denying No Pre-emption