-6

20 POINTS

1. 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 is a **global ring buffer** accessible to a **single producer** thread and **multiple consumer** threads (just as in assignment #3). You must write a solution that uses **event counters and sequencers (only if needed)** using the format shown:

The following types and operations are available:

```
ec_t   ec_name;              /* declare EC initialized with 0   */
seq_t  seq_name;             /* declare SEQ initialized with 0  */
void   await (ec_t * , int);    /*  await event                 */
void   advance (ec_t *);        /*  advance EC                  */
int    ticket (seq_t *);        /*  get a SEQ ticket            */
```

You must declare however many **event counters** and **sequencers** you need to solve this problem efficiently. The shared ring buffer is an array of **10 integer locations**. The single producer thread 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 threads must each 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**, make the necessary **EC** and **SEQ** (if needed) declarations in the box below and then write the **producer function** and the **consumer function** as described above.

GLOBAL TO PRODUCER AND CONSUMER THREADS:

DECALRE YOUR EC(s) AND SEQ(s) (IF NEEDED) HERE:

```
ec_t pEC, cEC;
seq_t cSEQ;
int in, out;
```

ALREADY DECLARED GLOBAL OBJECTS SHOWN BELOW:

```
int        ring buf[10];
```

WRITE PRODUCER FUNCTION HERE

```
void Producer(){
 while(1){
   await(&pEC,(in-10+1));
   buf[in] = random();
   ++in;
   advance(&cEC);
 }
}
```

WRITE CONSUMER FUNCTION HERE

```
Void consumer {
 int val; t
 while(1){ await(pec, t)
   await(&cEC, ticket(&cSEQ));
                t+1
   val = buf[out];
              t
   //print val here
   advance(&pEC);
 }
}
```

-4

15 POINTS

2. The following implementation of Peterson's synchronization algorithm for **2 threads** is **incorrect as shown**. The code is shown with line numbers for your reference. Each of 2 threads (**thread 0 and thread 1**) will repeatedly try to enter its critical section, do its critical section and then leave its critical section some arbitrary number of times. The code is incorrect in one of the lines shown, but is otherwise OK.

No extra lines are present, and no additional lines need to be added, but the incorrect line has to be fixed.

```
#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;                    // other process number
1  other = 1 - process;          // the opposite of process
2  interested[process] = TRUE;
3  turn = process;
4  while(interested[other] == TRUE || turn == other); // spin
}

// call enter_region(process_value) where process_value is 1 or 0
// execute critical section
// call leave_region(process_value) where process_value is 1 or 0

void leave_region(int process)  // process is 0 or 1
{
5  interested[process] = FALSE;  // drop my interested flag now
}
```

A. **Which line** is incorrect in the given code ?

line 4 ✓

B. **How** can this line cause the algorithm to fail ??

The while loop will not spin, as turn != other, and will never equal other.

no BW

mutex, Progress, bounded wait ? ? ?

C. Show how you would **re-code the line** to make the algorithm work Correctly

while(interest[other] == TRUE && turn = process);

15 POINTS

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

   A. 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 ?

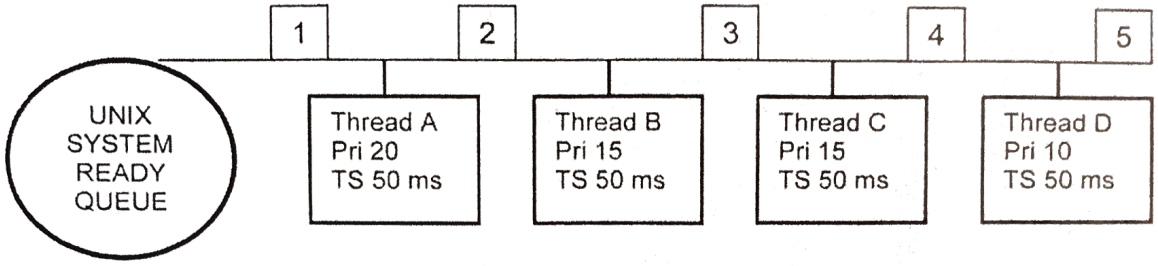
Real-time threads do not go through the aging process ✓

   B. Threads which are in the **RUN** state can transition off of the RUN state if they are: **EXITING**, being **PRIORITY PREEMPTED**, calling a **BLOCKING** operation or have expired their **TIME SLICE**. Assume that the queue depicted on the next page is the **READY** queue for a system which uses the **TS HPF/RR** scheduling policy for threads shown. Each element on the queue represents a thread waiting to execute with its priority **(Pri)** and time-slice **(TS)** value shown. Positions **between** the currently enqueued elements are numbered for reference, and in each of the following questions you must use **these numbers** to indicate where an element will be placed on the queue. You must indicate what each of the elements itemized below would look like and where each of these elements would go when they were placed on the READY queue shown. Assume high priority numbers represent best scheduling, and that the queue head is on the left. **Treat each element separately**, assuming that the queue looks just as it is shown at the time **each element** is considered (in other words, once you've determined where an element will go, forget it and consider the next element for the queue in its original form … elements **don't actually get added here**, we just want to know where they **would go** if they did).

PROBLEM 3 CONTINUED ON THE NEXT PAGE:

-2

PROBLEM 3 PART B CONTINUED:



FILL IN THE TS VALUE AND QUEUE POITION BELOW.
THE DEFAULT TIME SLICE FOR THIS SYSTEM IS 50 ms

1. This element is coming onto the ready queue after using up its TS

   Thread E
   Pri 20
   TS 50

   What position will it be placed at ?? 2

2. This element is coming onto the ready queue after returning from a blocking operation.

   Thread F
   Pri 10
   TS 50

   What position will it be placed at ?? ~~4~~ 5

3. This element is coming onto the ready queue after executing for 15 ms and being priority preempted.

   Thread G
   Pri 20
   TS 35

   What position will it be placed at ?? 1

4. This element is coming onto the ready queue as a new thread which was just created

   Thread H
   Pri 15
   TS 50

   What position will it be placed at ?? ~~3~~ 4

✓

-8

20 POINTS

4. Both the **Linux/UNIX** and **Windows** operating systems require that **some unique thread** be in the **RUN** state for each **processor (core)** in the system at all times.

   A. When a Linux/UNIX or Windows thread is **running in user mode** it is constrained to its own private address space. **All threads**, from time to time however, **must leave** their address spaces and **execute kernel code** in the kernel's address space. In **what ways** do threads leave their private address space and execute in the **kernel's address space** ??

   if they are signaled
   syscalls + exceptions

   B. The following simple program named `proc_run` takes an integer argument and will run on a Linux system when started from a shell prompt as shown:

```
-bash-3.00$  proc_run  1
```

```
int main(int argc, char* argv[]){
        int arg;
        char nstr[10];

        printf("LEVEL: %s\n", argv[1]);
        arg = atoi(argv[1]);
        if (++arg == 5)return;
        sprintf(nstr, "%d", arg);

        switch(fork()){
         case -1:
           exit(0);
         case 0:
           execl("./proc_run", "proc_run", nstr, NULL);
        }
        wait(NULL);
        printf("LEVEL: %s\n", argv[1]);
        return;
}
```

Write the exact output that the program will generate when it runs:

LEVEL : 1
LEVEL : 2
LEVEL : 3
LEVEL : 4
LEVEL : ~~4~~ 3
11 2
11 1

-6

15 POINTS

5. In class we discussed a synchronization example called the **reader/writer problem**. Any number or reader and writer threads may be interested in some common data, and while we can allow many readers to simultaneously access this data, any writer thread must access this data in pure mutual exclusion with all other reader and writer threads. You must write a solution to this problem using semaphores to implement a **reader_check_in()** function and a **reader_check_out()** function, as well as a single **writer()** function.

The following type, declaration, and operations are available:

```
sem_t sem_id = initial_integer_sem_value;
void  p( sem_t * );       /* this is a semaphore wait op   */
void  v( sem_t * );       /* this is a semaphore signal op */
```

Show the declaration and initialization of the semaphore(s) you need and any global variables that will be shared by either readers, writers or both.

SEMs and GLOBALS TO READERS AND WRITERS:

```
sem_t rsem; =1
sem_t wsem; =1
int reader=0;
```

WRITE THE reader_ check_in() HERE

```
void reader_check_in() {
while(1){
   P(&rsem);
   ++reader
   if(reader==0)
     P(&wsem);
   V(&rsem);
}}
```

WRITE THE reader_check_out() HERE

```
void reader_check_out(){
~~while(1){~~   P(rsem)
  --reader;
  if(reader==0)
     V(&wsem);
  V(&rsem);
}}
```

WRITE THE writer() HERE

```
void writer() {
   while(1){
     P(&wsem);
   //write code goes here
     V(&wsem);
}
```

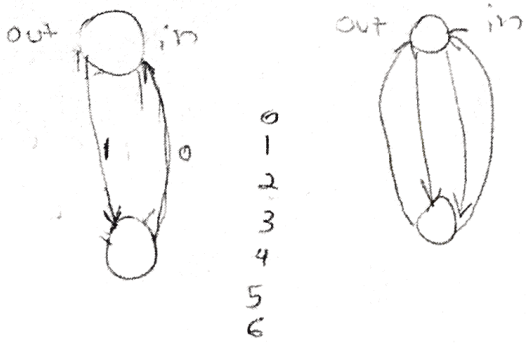
15 POINTS

6. The following complete program shows a **parent process** creating **two pipes** and then **forking a child process** which will run the **sort utility** just as you did in **assignment #2**. The parent reads the **same data** file used in assignment #2 and wants the child to sort it on the **first field** (last name) as **primary** key, and the **second field** (first name) as the **secondary** key. The parent will then read back the sorted data from the child and write it to the standard output (screen). (Assume all necessary include files are available, line numbers are for your reference)

```
1  int     main(int argc, char *argv[])
2  {
3          int     pfdout[2], pfdin[2], nread, a;
4          char    buf[81];
5
6          if(pipe(pfdout) == -1 || pipe(pfdin) == -1)
7          {
8                  perror("pipe");
9                  exit(1);
10         }
11
12         switch(fork())
13         {
14         case -1:  perror("fork");
15                   exit(2);
16
17         case  0:  close(0), close(1);
18                   dup(pfdin[1]);
19                   dup(pfdout[0]);
20                   close(pfdout[0]), close(pfdout[1]),
21                   close(pfdin[0]), close(pfdin[1]);
22                   execlp("sort", "sort", "-k 1,2", NULL);
23                   perror("execlp");
24                   exit(1);
25         } // end switch
26
27         close(pfdout[0]), close(pfdin[1]);
28
29         a=open("cs308a2_sort_data", O_RDONLY,0);
30         while(nread = read(a, buf, 80))
31                 write(pfdout[1], buf, nread);
32         close(pfdout[1]);
33         while(nread = read(pfdin[0], buf, 80))
34                 write(1, buf, nread);
35         return 0;
36 } // end main
```

stdout is closed first

stdin is closed after



Problem 6 continued next page:

Problem 6 continued:

A. In the above example, even though there are **NO syntax errors** and **NO system call errors,** the sort program takes a fatal error before any data can be processed (and the parent finishes without error, but without writing any sorted results). **Explain** why the sort program fails, and show **where** (using line numbers) and **what code changes** are necessary **for the parent and child to run as intended.**

The pipes on the child are switched. on line 17, stdout is closed first, then stdin. Then, on line 18, the in pipe is duped before the out pipe. To fix this, change the orderings of the dup()s. ✓

B. When a **Linux/UNIX system call** that returns a **channel number** such as **dup(), open()** or **socket()** is called, if the call succeeds we know that we will get a non-negative value returned. What else do we know about the actual value of the **returned channel number** ? The number returned was the lowest available channel number. ✓

The