20 POINTS

1. **Eventcounters** and **sequencers**, as described by Kanodia and Reed, provide somewhat of a decomposition of functionality when compared to **semaphores**. Whereas semaphores provide for **total ordering** of events, eventcounters alone can provide **partial ordering** of events, which is adequate for some types of synchronization problems, and, **in combination with sequencers**, can provide total ordering if needed. Below is a C style coded example of a **single producer, single consumer** problem which is solved with only eventcounters. You must rewrite the code so that it will support an efficient **multiple producer** and **multiple consumer solution**. Declare **any additional global objects** (beyond those already given below) required for your solution in the declaration box provided, and write the new producer and consumer code in the appropriate boxes.

BASIC GLOBALS TO BOTH

```
#define     N   100
ec_t        Pin, Cout;
int         buffer[N];
```

SINGLE PRODUCER

```
int     i = 0;
while(1){
  await(&Pin, i - N + 1);
  buffer[i % N] = make_donut();
  i++;
  advance(&Cout);
}
```

SINGLE CONSUMER

```
int     i = 0, donut;
while(1){
await(&Cout, i + 1);
  donut = buffer[i % N];
  i++;
    advance(&Pin);
}
```

The following types and operations are available for your solution:

```
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
```

Problem 1 is continued on next page:

-10

Problem 1 continued:

GLOBALS ALREADY DECLARED:

```
#define        N   100
ec_t           Pin, Cout;
int            buffer[N];
```

ADDITIONAL GLOBALS FOR MULTIPLE PRODUCERS / CONSUMERS

~~ec_t Pin B, Cout B;~~

need 2 sequencers

WRITE PRODUCERS' CODE HERE

```
int i=0;
void producer(){
  while(1){
    await(&Pin, i-N+1);
    await(&Pin B, i-N+1);
    buffer[i%N] = make_donut();
    i++;
    advance(&Cout);
    advance(&Cout B);
  }
}
```

WRITE CONSUMERS' CODE HERE

```
int i=0, donut;
void consumer(){
  while(1){
    await(&Cout, i+1);
    await(&Cout B, i+1);
    donut = buffer[i%N];
    i++;
    advance(&Pin)
    advance(&Pin B)
  }
}
```

-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 and enter its critical section, do its critical section and then leave its critical section some arbitrary number of times. The code has a flaw 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

int turn;
int interested[N];

void enter_cs(int thread){
      int other;

1.    other = 1 - thread;          // argument thread is a 1 or a 0
2.    interested[thread] = TRUE;
3.    turn = thread;
4.    while(interested[other] == TRUE && turn == other); //spin
}

        ....do critical section....

void leave_cs(int thread){
5.      interested[thread] = FALSE;
}
```

thread - 5

A. Which line is incorrect in the given code ?

4

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

because other is the opposite of turn/thread

Turn will never equal other, so the while loop will never spin. It has a ~~bound waiting issue~~.

no mutex !!

Both can be in critical section at the same time.

No mutex guarantee.

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

while (interested[other] == TRUE && turn == thread);

-3

15 POINTS

3. A **priority inversion** can occur among a collection of time sharing threads when a **low priority thread** obtains a mutually exclusive resource (like a lock of some kind) and a **medium priority thread** becomes **compute bound** and uses all available CPU cycles. When a **high priority thread** comes along that needs that same lock, it will have to **block** until the lock is free. Essentially, the **medium priority thread is blocking the high priority thread** (a priority inversion) because it never gives the **low priority thread** a chance to run and free the lock it holds. In **Windows and Linux/UNIX** systems, the kernel will **eventually resolve** these situations among time sharing threads.

A. Explain what **mechanism** the kernel will use to eventually **rescue** a priority inversion situation among time sharing threads.

It will dynamically allocate the correct times? The processes need to run via aging. ✓
Priority aging will boost low priority thread

B. If this situation occurs among **real time threads**, will the kernel intervene as with time sharing threads ? **Explain**.

NO Real time threads are not priority aged
the kernel treats real time threads in a hands off manner. The kernel will never intervene in real time threads

C. In a **single CPU** system, we can assume that the current thread in the **RUN** state on that CPU has a **priority that is equal to or greater than all other threads** that are in the READY state. **What aspect** (feature) of a **time sharing scheduling policy** allows that single CPU to be **shared** among some set of threads that **share the same highest priority** in the READY queue ? **Explain**.

The round robin feature allows threads w/ identical priorities to share CPU
Time share threads are scheduled with a time-slice (quantum), which forces a preemption with same priority peers when the TS expires
time slice / quantum to enable RR

-2

20 POINTS

4. . Both the **UNIX** and the **WindowsXP** operating systems require that **some unique thread** be in the **RUN** state for each **processor** in the system at all times.

A. When a UNIX or WindowsXP 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 the thread calls a sys call or runs into an exception ✓

Threads make system calls or handle exceptions on their CPUs

B. The following simple program named **forker** will run on a Linux system when started from the working directory it resides in, with a shell prompt as shown:

```
-bash-3.00$  ./forker
```

```
void fun(int);
int main(int argc, char* argv[]){
        printf("Calling fun now\n");
        fun(3);
        printf("I'm done\n");
        return;
}
void fun(int cnt){
        if(cnt){
           switch(fork()){
              case -1:
                 exit(0);
              case 0:
                 printf("cnt = %d\n", cnt);
                 fun(cnt - 1);
              default:
                 wait(NULL);
           }
        }
        return;
}
```

Cnt not Sc -1

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

Calling fun now

Cnt = 3

Cnt = 2 ✓

Cnt = 1

~~Cnt = 0~~

I'm done

I'm done

I'm done ✓

I'm done

15
60
900

-3

15 POINTS

5. In class we discussed a synchronization example called the **observer/reporter problem**. A run-forever **observer thread** detected cars as they passed by and wanted to increment a **shared global counter** for each passing car. A run-forever reporter thread mostly sleeps, but every so often it **awakens**, sends **the current car count** found in the global shared counter out to a printer, and then **resets the global shared counter to 0**. While either the reporter or the observer is using the shared counter the other thread must be kept away from the counter to avoid corrupting it.

Write code in 'C' style (C comments may be used for statements like *// print something someplace* or *// wait for car*) in the space provided below for **both the observer and reporter** thread functions with prototypes:

```
void  *observer (void *);    // a thread function with a while(1) loop
void  *reporter (void *);    // a thread function with a while(1) loop
```

using the **fewest number of semaphores 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 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 both the observer and the reporter in the box, (with initialization where needed), and then code each of the observer and reporter functions.

GLOBALS TO OBSERVER AND REPORTER:

```
sem_t lock = 1;      sem_t my_sem = 1;
int   counter = 0;   int   car_counter = 0;
```

WRITE OBSERVER FUNCTION HERE

```
void observer {
  while (1) {          when?
    p(&sem_id)
    //observe stuff
    counter ++;
    v(&sem_id)
  }

void observer (void) {
} while (1) {
    //when a car passes
    p(&my_sem);
    car_counter++;
    v(&my_sem);
  }// while
} // observer
```

WRITE REPORTER FUNCTION HERE

```
void reporter {
  while (1) {
    sleep(900)
    p(&sem_id)
    //prints obs count
    counter = 0;
    v(&sem_id)
} void reporter (void) {
    while (1) {
      sleep (900);
      p(&my_sem);
      //Print car_counter
      car_counter = 0;
      v(&my_sem);
    }//while
} // reporter
```

15 POINTS

6. The following complete program shows a parent process creating **2 pipes, and 2 children.** As you can see, one child is programmed to run the **ls –l** command and put its standard output into a pipe. The other child will run the **grep "Feb 16"** command against the **ls** generated data that it reads from that pipe, looking for **ls** lines that show a modification date of **Feb 16**, and then sending these lines into a second pipe that the parent will read. The parent will read the **grep** child's data from the second pipe and write it all out to the screen (the parent's stdout). (Assume all necessary include files are available; line numbers are for your reference)

```
1  int main(void){
2       int pchanA[2], pchanB[2], pidA, pidB, nread;
3       char mybuf[100];
4       if(pipe(pchanA) == -1 || pipe(pchanB) == -1){
5         perror("pipe");
6         exit(1);
7       }
8       switch(pidA = fork()){
9         case -1: perror("fork");
10                 exit(2);
11        case  0: close(1);
12                 if(dup(pchanA[1]) != 1){
13                      perror("dup pchanA ");
14                      exit(3);
15                 }
16                 execlp("ls", "ls", "-l", NULL);
17                 perror("exec ls ");
18                 exit(3);
19      } // switch
20      switch(pidB = fork()){
21        case -1: perror("fork");
22                 exit(2);
23        case  0: close(0);
24                 if(dup(pchanA[0]) != 0){
25                      perror("dup pchanA ");
26                      exit(3);
27                 }
28                 close(1);
29                   if(dup(pchanB[1]) != 1){
30                      perror("dup pchanB ");
31                      exit(3);
32                 }
33                 execlp("grep", "grep", "Feb 16", NULL);
34                 perror("exec grep ");
35                 exit(3);
36        default: close(pchanA[0]), close(pchanA[1]), close(pchanB[1]);
37                 printf("\nFiles modified on Jan 23 are: \n");
38                 while(nread = read(pchanB[0], mybuf, 80)){
39                      write(1, mybuf, nread);
40                 }
41      } // switch
42      return;
43 } // main
```

child 1 – ls -l
input ←
output → pipe

child 2
input – pipe from c1
output –

close (pchanA[1])

Feb 16th ☺

Problem #6 continued next page:

Problem #6 continued:

-7

A. In the above 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 **where** (using line numbers) and **what code changes** are necessary **for the parent to complete.**

Fix deadlock

The first child's input was never closed

It will not complete because the process will be stuck waiting for a EOF signal from the first child

I would add close(0); on line 11. inside the first child's switch statement

close pfhanA[1] before line 33

B. When a **system call** that returns a **channel number** such as **dup(), open()** or **socket()** is called, if the call succeeds, what do we know about the **returned channel number's numeric value ?**

The returned channel # is the lowest channel number it can possibly be. ✓

These calls will always return the smallest free channel in the channel table