-2

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 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 == process); // 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 ??

mutex and ~~bounded waiting or fail~~

no progress

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

while (interested [process] == TRUE && turn == process) ✓

-7

15 POINTS

6. The following complete program shows a parent process creating **a single pipe and child.** As you can see, the child is programmed to run the `ls -l` command and redirect its standard output to the pipe. The parent will read the child's output from the pipe and write it all out to a local file that the parent has opened for this purpose. (Assume all necessary include files are available, line numbers are for your reference)

```
1.  int main(void){
2.     int pchan[2], pid, nread, file_channel;
3.     char buf[100];
4.     if(pipe(pchan) == -1){
5.       perror("pipe");
6.       exit(1);
7.     }
8.     switch( pid = fork() ){
9.       case -1: perror("fork");
10.               exit(2);
11.      case  0: close(1);
12.               if( dup(pchan[1] ) != 1 ){
13.                   perror("dup");
14.                   exit(3);
15.               }
16.               execlp( "ls", "ls", "-l", NULL );
17.               perror("exec");
18.               exit(4);
19.      default: if((file_channel =
20.                    open("/tmp/data", O_CREAT|O_WRONLY, 0600)) == -1){
21.                   perror("open");
22.                   exit(5);
23.               }
24.               while(nread =  read(pchan[0], buf, 100)){
25.                   write(file_channel, buf, nread);
26.               }
27.               close(file_channel);
28.               if(close(pchan[1])== -1 || close(pchan[0])== -1){
29.                   perror("close");
30.                   exit(4);
31.               }
32.               wait(NULL);
33.               return(0);
34.    }
35. }
```

Parent must close(pchan[1]) before 24

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.**

line 11 the child never close().

close(Pchan[1])

execlp("ls", "ls", "-1", Null)

B. When a singly threaded process receives an **unblocked SIGSEGV** signal, we expect that the process **will terminate**, but there is a way that the thread in the process can take steps to **arrange for the process to continue executing**. **Explain** what steps the thread must take so it can continue executing even when an **unblocked SIGSEGV** is delivered to the process.

install signal handler

-12

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 are a set of **global objects** which are accessible to **<u>any number of</u> producer** threads and **<u>any number of</u> consumer** threads running in a single process. 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. Declare and initialize however many semaphores you need to solve this problem **efficiently**. The shared ring buffer is an array of 10 **integer locations**. Each 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**. Each consumer must execute a **forever loop** taking a number out of the shared ring buffer and printing it 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()** for wait and **v()** for signal (prototype headers are declared below the global data). **Busy-waiting** is not allowed anywhere in your solution.

GLOBALS TO PRODUCER AND CONSUMER THREADS:

```
sem_t sem_name = sem_initial_value;
```

← **<u>format</u>**

DECALRE YOUR SEMAPHORES HERE

```
sem_t Psem = 1
sem_t Csem = 1
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

```
Void Prod ( ) {
  int i;
forever



}
```

WRITE CONSUMER FUNCTION HERE

```
Void Cons( ) {
    int i = 0; domt;
  forever;



}
```

-7

15 POINTS

5. 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** — 900 sec between reports. The following types and operations are available:

```
ec_t  event_counter;             // declare an EC with value of 0
seq_t sequencer;                 // declare a SEQ with value of 0
void  await   (ec_t * , int);    // await event
void  advance (ec_t *);          // advance EC
int   ticket  (seq_t *);         // get 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) in the box below, and then code each of the observer and reporter functions.

GLOBALS TO OBSERVER AND REPORTER:

```
ec_t ec!
seq_t seq
int count = 0;
```

WRITE OBSERVER FUNCTION HERE

```
void *observer ( void x )
{ ~~int i = 0;~~
  while(1){        & ec      ticket(& seq)
    await(& event_counter, ~~900-i~~);
                            ^
    ticket(& sequencer);
    i++; count++
    advance (&event_counter)
  }
      etc

}
```

WRITE REPORTER FUNCTION HERE

```
void *reporter (void x)
{ int i = 0;
  while(1){
     await(& event_counter, 900-i);
     i++;

     advance(& event counter)

  }

}
```

-8

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 Linux/UNIX) the default scheduling policy for non-privileged threads is a time-sharing (**TS**) policy using an **HPF/RR** (Highest Priority First / Round Robin) dispatching mechanism.

   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 ?

   no aging

   B. A thread in a Linux system is either a **timesharing thread** or a **real time thread**. If the **absolute priority** of a particular Linux thread is **43**, **is it a timesharing thread or a real time thread ? Explain.**

   RT 40 - 139
   TS 0 - 39

   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 <u>aspect</u>** (feature) of a **<u>time sharing scheduling policy</u>** allows that single CPU to be **shared** among some set of threads that **have the same highest priority** in the READY queue ? **Explain.**

   time slice ✓ become the thread with the same priority have their order to determined by their timeslice space running.

−10

20 POINTS

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

A. If all available user threads are **blocked** at a time when the current running thread **enters the context switch to block as well**, how are these systems able to find a thread to run ??

it will swap with the thread that is equal to or greater thread priority.

idle thread

B. When a 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** (by what methods) do threads leave their private address space and execute in the **kernel's address space** ??

thread makes system call or ✓

hadley exception.

C. The **thread state diagram** we've been looking at in class is partially drawn below, using a **circle** for each **thread state** and **directed arcs** for each possible transition. The **RUN** state is labeled. You must <u>**label**</u> the other **two states** and <u>**label**</u> all of **the directed arcs.**