90

NAME Hoang Do

EXAM #1 COMP.3080 OPERATING SYSTEM
March 5, 2019

Print (or write very clearly) your answers in the space provided on the exam....if you need more space for your answer use the back of the exam sheet and indicate in the primary answer space that you have used the back.

There are 6 questions with points as shown for a total of 100 points....the exam will run for the full class period, but you can hand it in and leave whenever you're finished.

You do not have to pass in the reference pages, but if you have work on them that I should see, please include them with your exam submission, and make an appropriate notation on the exam itself to the work on the reference pages.

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:

Problem 8 continued:

GLOBALS ALREADY DECLARED:

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

ADDITIONAL GLOBALS FOR MULTIPLE PRODUCERS / CONSUMERS

seq_t pSEQ, cSEQ ✓

-6

WRITE PRODUCERS' CODE HERE

```
void producer {
  int t;
  while (1) {
    t = ticket(&pSEQ)
    await(&~~Pin~~ Cout, t);
    await(&Pin, t-~~10~~ N +1);
    buffer[t%N] = make_donut();
    advance(&cout);
  }
}
```

WRITE CONSUMERS' CODE HERE

```
void consumer {
  int u, donut;
  while (1) {
    u = ticket(&cSEQ)
    await(&~~Cout~~ Pin, u);
    await(&Cout, u+1);
    donut = buffer[u%N];
    //print donut somewhere
    advance(&Pin);
  }
}
```

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;
}
```

A. Which line is incorrect in the given code ?

4

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

turn == thread

No bounded waiting, no mutex

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

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

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.

Priority aging will boost the low thread's priority high enough to have a chance to run.

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

No, because real time threads are not priority aged

C. 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.**

- Range of priority:
  + timesharing thread : 0-39
  + Real time thread : 40-139

=) If the absolute priority of a particular Linux is 43, it is a Real time thread.

-4

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** ??

Threads leave their private address space and execute in Kernel's address space when it makes a system call or in exception handler

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(2);
        printf("fun done\n");
        return;
}
void fun(int cnt){
        if(cnt){
           switch(fork()){
             case -1:
                exit(0);
             case 0:
                printf("this is kid %d\n", cnt);
                fun(cnt - 1);
                break;
             default:
                wait(NULL);
                printf("kid %d done\n",cnt);
           }
        }
        return;
}
```

2 / 1

2.

pid A   pid B   pid C

fun(2) → fun(1) → fun(0)

finish A ← finish B ← finish C

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

Calling fun now

this is kid 2

kid 2 done

this is kid 1

kid 1 done

fun done

~~fun done~~

~~fun done~~

fun done

Calling fun now

this is kid 2

This is kid 1

fun done   C

kid 1 done

fun done   B

kid 2 done

fun done   A

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 other global variable(s) that will be shared by either readers, writers or both **in the box below**.

SEMs and GLOBALS TO READERS AND WRITERS:

sem_t rdS = 1, wrS = 1;

int nreader = 0; ✓

WRITE THE reader_ check_in() HERE
void reader_check_in() {

p = (& rdS) ;
if ~~while~~ (nreader == 0)
p = (& wrS);
++ nreader ;
v = (& rdS);

} // Reader Can read

WRITE THE reader_check_out() HERE
void reader_check_out(){

p = (& rdS);
-- nreader ;
if while (nreader == 0)
v = (& wrS);
v = (& rdS) ;

}

WRITE THE writer() HERE
void writer() {

p = (& wrS) ;

v = (& wrS) ;

✓

}

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                            pfdout      pfdin
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
```

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.**

In line 17, pfdout is close first and after that pfdin, but the system call dup(pfdin[1]) (line 18) before dup (pfdout[0]) line 19.

So swap line 18 and 19 to correct the code ✓

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 system call will returns the lowest available number of the channel number. ✓