Exam 2 Comments

It takes a really bad school to ruin a good student and a really fantastic school to rescue a bad student.

Dennis J. Frailey
Write your answers in a technical/formal style.

Avoid the use of imprecise and non-professional wording and language as computer science is an exact science and we must learn to communicate in a professional way.

Present all key elements as grading is based on how many key elements are answered properly.

Justify your answer. For example, if you claim bounded violation is violated, then show it with an execution sequence. Don’t make a vague claim without a good justification.

I do not do grade inflation.
int flag[2]; // initialized to OUT_CS
int turn;    // initialized to 0 or 1

Process $i$ ($P_i$), $i = 0$ or 1
// Enter Protocol
repeat
    flag[i] = REQUEST;
    while (turn != i && flag[j] != OUT_CS)
        ;
    flag[i] = IN_CS;
until flag[j] != IN_CS;
turn = i;

Critical Section
// Exit Protocol
turn = j;
flag[i] = OUT_CS;

- Process $P_i$ exits its repeat-until loop sees flag[j] being not IN_CS, and right before that $P_i$ sets flag[i] to IN_CS.
- By the same reason, if $P_j$ is in its critical section, flag[j] is IN_CS and flag[i] is not IN_CS.
- If $P_i$ and $P_j$ were both in the critical section, flag[i] would be IN_CS and not IN_CS at the same time.
Variable turn is not used in the reasoning.

If $P_i$ and $P_j$ are both in the critical section, they execute the statement turn = $i$.

If $P_i$ executes this statement first followed by $P_j$, the value of turn is $j$.

If $P_j$ executes this statement first followed by $P_i$, the value of turn is $i$.

Hence, turn will not have two values at the same time.
Variable \textit{turn} is set only once in the \textit{if} statement and is not reset when exits the critical section.

Suppose \( P_0 \) sets \textit{turn} to 0 and enters the critical section. Because \( P_0 \) does not reset \textit{turn}, \( P_0 \) may come back and re-enter the critical section.

This may repeat again and again, and bounded waiting fails.
# Problem 1(b): 2/2

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_1$</td>
<td>turn</td>
<td>status[0]</td>
<td>status[1]</td>
<td>Comment</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$s[0]$=C</td>
<td>$s[1]$=C</td>
<td>0</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>while</td>
<td>while</td>
<td>0</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>$s[0]$=IN</td>
<td></td>
<td>0</td>
<td>IN</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>$s[1]$=OUT</td>
<td></td>
<td>0</td>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$s[0]$=OUT</td>
<td>if</td>
<td>0</td>
<td>OUT</td>
<td>OUT</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$s[0]$=C</td>
<td>while</td>
<td>loops back</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>10</td>
<td>if</td>
<td></td>
<td>0</td>
<td>C</td>
<td>OUT</td>
</tr>
<tr>
<td>11</td>
<td>while</td>
<td>while</td>
<td>0</td>
<td>C</td>
<td>OUT</td>
</tr>
<tr>
<td>12</td>
<td>$s[0]$=IN</td>
<td></td>
<td>0</td>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
status[0] = COMPETING;
do {
    while (turn != 0) {
        status[0] = OUT_CS;
        if (status[turn] == OUT_CS)
            turn = 0;
    }
    status[0] = IN_CS;
} while (status[1] == IN_CS);
```
A race condition is a situation in which more than one process or thread access a shared resource concurrently, and the result depends on the order of execution.

Use instruction level execution sequences for your examples.

You must show concurrent sharing in your execution sequences.

It takes two execution sequences to justify the existence of a race condition, because you need to show the results depend on the order of execution.
This is not a valid example to show the existence of a race condition because variable x is not shared concurrently.
Problem 1(c): 3/10

int Count = 10;

Process 1  Process 2

Count++; Count--;

Count = 9, 10 or 11?

Higher-level language statements are not atomic

Only say Count++ and Count-- would cause a race condition is inaccurate because the “sharing” and “concurrent access” conditions are not addressed.
Problem 1(c): 4/10

int Count = 10;

Process 1

LOAD Reg, Count
ADD #1
STORE Reg, Count

Process 2

LOAD Reg, Count
SUB #1
STORE Reg, Count

The problem is that the execution flow may be switched in the middle. Possible answers are 9, 10 or 11. Show two execution sequences.
## Problem 1(c): 5/10

### First Execution Sequence

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst</td>
<td>Inst</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td>Memory</td>
<td>Memory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOAD</th>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>ADD</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

overwrites the previous value 11

STORE 9

9
### Problem 1(c): 6/10

#### Second Execution Sequence

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inst</strong></td>
<td><strong>Inst</strong></td>
</tr>
<tr>
<td>LOAD</td>
<td>LOAD</td>
</tr>
<tr>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td>STORE</td>
<td>STORE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

The STORE instruction in Process 1 overwrites the previous value 9 in Process 2.
You should use instruction level interleaving to demonstrate the existence of race conditions, because

a) higher-level language statements are not atomic and can be switched in the middle of execution

b) instruction level interleaving can show clearly the “sharing” of a resource among processes and threads.
Problem 1(c): 8/10

```c
int a[3] = { 3, 4, 5};

Process 1                                 Process 2
```

**Execution Sequence 1**

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Array a [ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]=a[0]+a[1]</td>
<td></td>
<td>{3, 7, 5}</td>
</tr>
</tbody>
</table>

There is no concurrent sharing, not a valid example for a race condition.

**Execution Sequence 2**

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Array a [ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1]=a[0]+a[1]</td>
<td></td>
<td>{3, 7, 9}</td>
</tr>
</tbody>
</table>
Problem 1(c): 9/10

```c
int Count = 10;
```

### Process 1
- LOAD Reg, Count
- ADD #1
- STORE Reg, Count

### Process 2
- LOAD Reg, Count
- SUB #1
- STORE Reg, Count

---

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD Reg, Count</td>
<td>LOAD Reg, Count</td>
<td>10</td>
</tr>
<tr>
<td>SUB #1</td>
<td>SUB #1</td>
<td>10</td>
</tr>
<tr>
<td>ADD #1</td>
<td>STORE Reg, Count</td>
<td>10</td>
</tr>
<tr>
<td>STORE Reg, Count</td>
<td>STORE Reg, Count</td>
<td>11</td>
</tr>
</tbody>
</table>

variable count is shared concurrently here
The following execution sequence is not acceptable, because `count++` and `count--` are higher level language statements mixed with machine instructions. These statements apply to memory and have immediate impact.

```c
int count = 0;
```

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD count</td>
<td>LOAD count</td>
</tr>
<tr>
<td>count++</td>
<td>count--</td>
</tr>
<tr>
<td>SAVE count</td>
<td>SAVE count</td>
</tr>
</tbody>
</table>

The following is an obvious solution.

Semaphore $S1 = 1$, $S2 = 0$, $S3 = 0$;

**Thread 1**
```
while (1) {
    S1.Wait();
    cout << "1";
    S2.Signal();
}
```

**Thread 2**
```
while (1) {
    S2.Wait();
    cout << "2";
    S3.Signal();
}
```

**Thread 3**
```
while (1) {
    S3.Wait();
    cout << "3";
    S2.Signal();
    S2.Wait();
    cout << "2";
    S1.Signal();
    // do something
}
```
If you insist that **Thread 2** can only have one statement to print 2, here is another solution.

After printing 2 the first time, the printing process goes "**forward**" to Thread 3. Then, the next time, the printing process goes "**backward**".

```
semaphore S1 = 1, S2 = 0, S3 = 0;

Thread 1
while (1) {
    S1.Wait();
    cout << "1";
    S2.Signal();
}

Thread 2
int Forward = TRUE;
while (1) {
    S2.Wait();
    cout << "2";
    if (Forward)
        S3.Signal();
    else
        S1.Signal();
    Forward = !Forward;
}

Thread 3
while (1) {
    S3.Wait();
    cout << "3";
    S2.Signal();
}"
```
If `Wait()` is not atomic, multiple threads can call `Wait()` and increase the counter at the same time. Race condition can happen.

---

### Mutual Exclusion

- Semaphore `S=1;`
- `P_0` & `P_1` enter the critical section simultaneously if `S.Wait()` is not atomic.

### Critical Section

- `S.Wait();`

---

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>=1 for M.Ex</td>
</tr>
<tr>
<td>2</td>
<td>S.Wait()</td>
<td>S.Wait()</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>LA count</td>
<td>LA count</td>
<td>Reg=1, 1</td>
</tr>
<tr>
<td>4</td>
<td>SUB #1</td>
<td>SUB #1</td>
<td>Reg=0, 1</td>
</tr>
<tr>
<td>5</td>
<td>SA count</td>
<td>SA count</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Both processes enter their critical sections</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We assume the weirdo (philosopher 5) always picks his right chopstick first followed by his left one, and all normal ones pick their left first.

```c
if the weirdo has his right chopstick then
    if the weirdo has his left chopstick then
        the weirdo eats and there is no deadlock.
    else // weirdo’s left is taken by philosopher 4 as his right
        philosopher 4 eats. no deadlock.
else // the weirdo does not have his right because philosopher 1 has it as his left
    // weirdo’s left is available
    if philosopher 1 has his right then
        philosopher 1 eats and there is no deadlock
    else // philosopher 1’s right is taken by philosopher 2 as his left
        if philosopher 2 has his right then
            philosopher 2 eats and there is no deadlock
        else // philosopher 2’s right is taken by philosopher 3 as his left
            if philosopher 3 has his right then
                philosopher 3 eats and there is no deadlock
            else // philosopher 3’s right is taken by philosopher 4 as his left
                philosopher 4 eats as he can use weirdo’s left as his right
```
**Problem 2(c): 2/2**

**weirdo has his right**

*philosopher 1 cannot eat*

if weirdo also has his left, weirdo eats

if weirdo does not have his left, it is taken by philo 4 as his right and philo 4 eats

**Figure (a)**

**Figure (b)**

If weirdo does not have his right, this chop is taken by philo 1, .... Philo 4 eats

**Figure (c)**

**Figure (d)**
The following is the basic code:

```c
while (1) {
    a[i] = f(a[i], a[(i+1) % n]);
    Center = a[i] + Center;
}
```

Thus, thread $T_i$ uses $a[(i+1) \% n]$ and modifies $a[i]$ and Center.

This is similar to the dining philosophers problem.
We need a semaphore for each $a[i]$. $T_i$ needs two semaphores for $a[i]$ and $a[(i+1)\%n]$ to access $a[i]$ and $a[(i+1)\%n]$.

Because $Center$ is accessed by all threads, we also need a semaphore to protect $Center$.

```c
semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, ...., 1};
```
Problem 3(a): 3/5

semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, ….., 1};

while (1) {
    S_a[(i+1)%n].Wait();
    Local = a[(i+1)%n];
    S_a[(i+1)%n].Signal();
    fx = f(a[i], Local);
    S_a[i].Wait();
    a[i] = fx;
    S_a[i].Signal();
    S_Center.Wait();
    Center = fx + Center;
    S_Center.Signal();
}

Because \( f() \) does not modify \( a[i] \) and \( Local \), no lock needed.
semaphore S_Center = 1;
semaphore S_a[n] = { 1, 1, ..... , 1};

while (1) {
    // other irrelevant computation
    S_a[(i+1)%n].Wait();
    S_a[i].Wait();
    S_Center.Wait();
    a[i] = f(a[i], a[(i+1)%n]);
    Center = a[i] + Center;
    S_Center.Signal();
    S_a[i].Signal
    S_a[(i+1)%n].Signal
    // other irrelevant computation
}

This implementation serializes all threads, no concurrency at all. Only one thread can modify a[] (not OK) and Center (OK).
Problem 3(a): 5/5

semaphore S;

while (1) {
    // other irrelevant computation
    S.Wait();
    a[i] = f(a[i], a[(i+1)%n]);
    Center = a[i] + Center;
    S.Signal();
    // other irrelevant computation
}

This solution is even worse because there is no concurrency.
All men can use the bathroom as long as there is a man using it. Aren’t the man threads readers in the readers-writers problem?

By the same reason, all women can use the bathroom as long as there is a woman using it. Therefore, all woman threads form another “reader” threads in the readers-writers problem.

In conclusion, we have two groups of readers, and while one group of readers is using the bathroom the other group is blocked.

What we need? Duplicate the reader thread, one for men and the other for women.
Problem 3(b): 2/2

```c
int MaleCounter = 0, FemaleCounter = 0;
Semaphore MaleMutex = 1, FemaleMutex = 1;
Semaphore BathRoom = 1;

while (1) {
    // working
    MaleMutex.Wait();
    MaleCounter++;
    if (MaleCounter == 1)
        BathRoom.Wait();
    MaleMutex.Signal();
    // use the bathroom
    MaleMutex.Wait();
    MaleCounter--;
    if (MaleCounter == 0)
        BathRoom.Signal();
}

if I am the first man/woman,
yield the bathroom

if I am the last man/woman,
yield the bathroom
```
I expected you to receive approximately 70 points as shown below.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Expected</th>
<th>Class Average</th>
<th>Class Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>b</td>
<td>15</td>
<td>7</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>c*</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>15</td>
<td>10</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>69</td>
<td>53</td>
<td>56</td>
</tr>
</tbody>
</table>
### Grade Distribution

**Problem-Wise**

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1b</th>
<th>1c</th>
<th>2a</th>
<th>2b</th>
<th>2c</th>
<th>3a</th>
<th>3b</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>Median</td>
<td>13</td>
<td>0</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>11</td>
<td>56</td>
</tr>
<tr>
<td>Avg</td>
<td>10</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>53</td>
</tr>
<tr>
<td>St DEV</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>26</td>
</tr>
</tbody>
</table>

- Problem 1a is a problem similar to Attempt II
- Problem 1b is a little more difficult, but you have a hint
- Problem 1c is a “recycled” problem from EXAM I
- Problem 2a, 2b and 2c were exercises assigned in class
- Problem 3a is similar to the philosophers problem and 3b is a variation of the readers-writers problem.
Boxplot

1st quartile (29.5)  
median (56)  
3rd quartile (69.5)  
maximum (100)

minimum (0)  
average (53)

¼ scores  
¼ scores  
¼ scores  
¼ scores
Grade Distribution

- <30: 1
- ≥30: 7
- ≥50: 5
- ≥70: 3
- ≥60: 10
- ≥80: 11
- ≥90: 6

32
Many of you did not study the slides carefully. Even the easiest problems were answered poorly/incorrectly.

Some just provide an answer or value without elaboration. I am not supposed to finish your answer for you. Whenever a justification and/or elaboration is needed, please do it. Use correct wording.

Please study harder, ask questions, and make sure you understand the subjects.

Your grade is proportional to the quality of your answers and is not proportional to the time you spent!

In my experience the Final is usually easier because difficult topics are spread thin.

Again, I do not do grade inflation.
It takes a really bad school to ruin a good student and a really fantastic school to rescue a bad student.

Dennis J. Frailey
The End