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

- 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 there is a race condition, then show it with execution sequences.
- I do not do grade inflation.
Problem 1(a)i

- Your output should look like the table shown on the right side of this slide if you ran your program on an Intel-based CPU.
- From 0! To 20!, the results are correct.
- 21!, 22! and 24! become negative and 25! is less than 23!
Problem 1(a)ii: 1/3

- The minimum and maximum of `long` are system dependent. See `limits.h` for the details.

- On my MacBook Air and iMac under `gcc`, the minimum and maximum values of the `long int` type, which are the same as the `long long int` type, are \(-9223372036854775808\) and \(9223372036854775807\).

- In general, if a signed integer is represented by \(k+1\) bits with 1 sign bit, then the minimum and maximum are likely to be \(-2^k-1\) and \(2^k-1\). If the computed result is larger than \(2^k-1\), only the last \(k\) bits would be stored.
Problem 1(a)ii: 2/3

- Suppose a 4-bit register is used for multiplication.
- Multiplying $0110_2 = 6_{10}$ and $0101_2 = 5_{10}$ yields $30_{10} = 11110_2$.
- Because we only use 4-bit registers, the stored result would be the last 4 bits $1110_2$.
- Because we use signed integers, the first bit is the sign bit (i.e., 0 – positive and 1 – negative), and $1110_2$ actually means $-2_{10}$ under the commonly seen 2's complement system. Note that different computer architectures would produce different results.
Problem 1(a)ii: 3/3

- 4-bit Representation

2’s complement:

1110₂ (negative)
0001₂ (bit flipping)
0010₂ (adding 1)
-2₁₀

MAX + 1 → MIN

wrapping

0000₂ = 0₁₀
0001₂ = 1₁₀
0010₂ = 2₁₀
0011₂ = 3₁₀
0100₂ = 4₁₀
0101₂ = 5₁₀
0110₂ = 6₁₀
0111₂ = 7₁₀
1000₂ = -8₁₀
1001₂ = -7₁₀
1010₂ = -6₁₀
1011₂ = -5₁₀
1100₂ = -4₁₀
1101₂ = -3₁₀
1110₂ = -2₁₀
1111₂ = -1₁₀
Modern CPUs have two execution modes: the user mode and the supervisor (or system, kernel, privileged) mode, controlled by a mode bit.

The OS runs in the supervisor mode and all user programs run in the user mode. Some instructions that may do harm to the OS (e.g., I/O and CPU mode change) are privileged instructions, which, for most cases, can only be used in the supervisor mode.

When execution switches to the OS (resp., a user program), execution mode must be changed to the supervisor (resp., user) mode.
An **interrupt** is an event that requires OS’s attention. It may be generated by hardware (e.g., I/O completion and timer) or software (e.g., system call and division by 0).

Interrupts generated by software (e.g., division by 0, page fault and system call) are traps.

Don’t forget mode switch.

Interrupts are **not signals** and are **not called**. Signals have a different meaning in operating systems.
Interrupts are not machine instructions, not signals, not functions/procedures.

Signals have a different meaning in OS.

Interrupts, machine instructions, threads, processes are **NOT** called.

OS does not call an interrupt. Except for system calls and a few others, interrupts are not called to happen.

Many answered this question by stating the result of an interrupt rather than talking about an interrupt itself.
Problem 3(a)

- **new**
- **ready**
- **running**
- **waiting**
- **terminated**

**Transitions:**
- Converting to process
- Scheduler dispatch
- Interrupt
- Waiting for CPU
- Waiting for I/O or event
- I/O or event completion
- I/O or event wait
- Exit

**Actions:**
- Admitted
- Reclaim resource
- Destroy process

**States:**
- New
- Ready
- Running
- Waiting
- Terminated
The context of a process is the environment for that process to run properly.

This includes process ID, process state, registers, memory areas, program counter, files, scheduling priority, etc.

The sequence of actions are:

- Control switches back to the OS. Mode switch may be needed.
- The outgoing process is suspended, and its context saved. Depending on the nature of this context switch, this outgoing process may be moved to the Ready or Waiting state. It could also be moved to the Terminated state if it exits or causes an error.
- The context of the incoming process is loaded and its state is set to Run.
- Resume its execution. Mode switch may be needed.
A race condition is a situation in which more than one processes or threads 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.
int Count = 10;

Process 1

Count++; Count--;

Process 2

Count = 9, 10 or 11?

Only say `Count++` and `Count--` would cause a race condition is inaccurate because the “sharing” and “concurrent access” conditions are not addressed.

Higher-level language statements are not atomic.
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 4(a): 5/9

**First Execution Sequence**

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>10</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>

<table>
<thead>
<tr>
<th>Inst</th>
<th>Reg</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SUB</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

*overwrites the previous value 11*
### 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><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>STORE</td>
<td>STORE</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

The STORE instruction overwrites the previous value **9**.
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 4(a): 8/9

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

Process 1                                 Process 2
```

```c
```

---

**Execution Sequence 1**

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Array a[ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a[1]=a[0]+a[1]</code></td>
<td></td>
<td><code>{3, 7, 5}</code></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><code>a[1]=a[0]+a[1]</code></td>
<td></td>
<td><code>{3, 7, 9}</code></td>
</tr>
</tbody>
</table>
**Problem 4(a): 9/9**

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

<table>
<thead>
<tr>
<th><strong>Process 1</strong></th>
<th><strong>Process 2</strong></th>
<th><strong>Memory</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD Reg, Count</td>
<td>LOAD Reg, Count</td>
<td>10</td>
</tr>
<tr>
<td>ADD #1</td>
<td>SUB #1</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.
Problem 5(a)

printf("The root process %d, ppid = %d\n\n", getpid(), getppid());
n = atoi(argv[1]);
for (i = 1; i <= n; i++) {
    if ((pid = fork()) == 0) {
        // left child
        printf("My ID = %d My PPID = %d\n\n", getpid(), getppid());
        exit(0);
    // left child must exit
    } else {
        // parent
        if ((pid = fork()) == 0) {
            // right child
            printf("My ID = %d My PPID = %d\n\n", getpid(), getppid());
        // must keeps creating
        } else {
            // parent must wait for
            wait(NULL);
            wait(NULL);
            // both children
            exit(0);
        // parent exits
    }
}
}
Obvious cases are as follows (i.e., 2, 3 and 4):

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>x in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = 2*x</td>
<td>0</td>
</tr>
<tr>
<td>x++</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>x++</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>x in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>x = 2*x</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>x++</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

The final result cannot be greater than 4, because \( x = 2 \times x \) can only double the result of Process 1.
Non-obvious cases are as follows (i.e., 0 and 1):

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>x in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOAD ( x )</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MUL #2</td>
<td>0</td>
</tr>
<tr>
<td>( x++ )</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( x++ )</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>SAVE ( x )</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>x in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOAD ( x )</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MUL #2</td>
<td>0</td>
</tr>
<tr>
<td>( x++ )</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>( x++ )</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SAVE ( x )</td>
<td>0</td>
</tr>
</tbody>
</table>
Problem 5(c): 1/2

```c
int status[2];    // status of a process
int turn;         // initialized to either 0 or 1

P_0
status[0] = COMPETING;
while (status[1] == COMPETING) {
    status[0] = OUT_CS;
    repeat until (turn == 0);
    turn = 0;
    status[0] = COMPETING;
}

P_1
status[1] = COMPETING;
while (status[0] == COMPETING) {
    status[1] = OUT_CS;
    repeat until (turn == 0 || turn == 1);
    turn = 1;
    status[1] = COMPETING;
}
```

Before entering `while`, `status[ ]` is COMPETING
When loops back `status[ ]` is set to COMPETING

\( P_0 \) enters its critical section
\( \text{iff } \) `status[0]` is COMPETING and `status[1]` not COMPETING

\( P_1 \) enters its critical section
\( \text{iff } \) `status[1]` is COMPETING and `status[0]` not COMPETING

If both \( P_0 \) and \( P_1 \) are in their critical section, `status[0]` (and `status[1]`) must be COMPETING and not COMPETING at the same time.
Problem 5(c): 2/2

```c
int status[2]; // status of a process
int turn;     // initialized to either 0 or 1

P_0
status[0]=COMPETING;
while (status[1]==COMPETING) {
    status[0]=OUT_CS;
    repeat until (turn==0);
    turn = 0;
    status[0] = COMPETING;
}

P_1
status[1] = COMPETING;
while (status[0]==COMPETING) {
    status[1]=OUT_CS;
    repeat until (turn==0 || turn==1);
    turn = 1;
    status[1] = COMPETING;
}
```

turn plays no role here.

**REASON:**

If a process sets `status[ ]` to COMPETING and finds the other `status[ ]` being non-COMPETING, this process enters its critical section.

In this case, the process never sets its `status[ ]` to OUT_CS and turn to 0 or 1.

Hence, you should not use turn and OUT_CS in your argument.
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>a(i)</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3(ii)</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>10</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>10</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>15</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>15</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>74</td>
<td>55</td>
<td>51</td>
</tr>
</tbody>
</table>

50 points expected from class slides directly
### Grade Distribution

#### Problem-Wise

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1b</th>
<th>2a</th>
<th>2b</th>
<th>3a</th>
<th>3b</th>
<th>4a</th>
<th>5a</th>
<th>5b</th>
<th>5c</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Min</strong></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>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>92</td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td><strong>St DEV</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>

- Problems 2a, 2b, 3a, 3b and 4a are from our course slides.
- Problem 1 is an exercise stated in Programming Assignment I.
- Problem 5a tests whether you know `fork()` properly.
- Problem 5b tests whether you can use machine instruction interleaving.
- Problem 5c is a simple problem using prove-by-contradiction.
Boxplot

Average > Mean →
top half performed better than the lower half

1st quartile (42)
2nd quartile (median) (51)
3rd quartile (69)
average (55)
minimum (6)
maximum (92)

There were no outliers

50% of your scores is in the range of 42 and 69

Average > Mean →
top half performed better than the lower half

1st quartile (42)
2nd quartile (median) (51)
3rd quartile (69)
average (55)
minimum (6)
maximum (92)
If the 12 scores lower than 40 are removed, class average and median become 61 and 59.
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.**

If execution sequences are needed, always provide valid ones. Otherwise, you will receive a **ZERO**.

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!

**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