PROBLEM 1

Question 1.1

[A direct translation of the CSP program to a Petri Net is readily made. It may, however, be recognized that between rounds, $P_2$ has to synchronize with both $P_1$ and $P_3$ (in any order). This can succinctly be expressed by:]

\[ A \quad B \quad C \]

Question 1.2

As $A$ and $C$ both (and independently) have to synchronize with $B$ for each execution, the following predicate is a characteristic invariant of the program:

\[ I \triangleq |a - c| \leq 2 \]

Question 1.3

The above Petri Net can be implemented directly with semaphores:

```plaintext
var S_A, S_BA, S_BC, S_C : semaphore;
S_A := 0; S_BA := 0; S_BC := 0; S_C := 0;

process P_A;
repeat
A;
P(S_A);
V(S_BA);
forever
process P_B;
repeat
B;
V(S_A);
V(S_C);
P(S_BA);
P(S_BC);
forever
process P_C;
repeat
C;
P(S_C);
V(S_BC);
forever
```

[The $P$ and $V$ operations in $P_A$ and $P_C$ might be swapped. With the given ordering though, the semaphores $S_{BA}$ and $S_{BC}$ may be replaced by a common semaphore.]
PROBLEM 2

Question 2.1

(a) Transition diagrams:

\[ P_1: \]
\[
\begin{array}{c}
(0) \\
(1) \\
(2) \\
(3) \\
(4)
\end{array}
\]
\[
\begin{array}{c}
a_1: t_1 := y \\
a_2: y := t_1 + 4
\end{array}
\]

\[ P_2: \]
\[
\begin{array}{c}
(0) \\
(1) \\
(2) \\
(3) \\
(4)
\end{array}
\]
\[
\begin{array}{c}
b_1: x := x + 1
\end{array}
\]

\[ P_3: \]
\[
\begin{array}{c}
(0) \\
(1) \\
(2) \\
(3) \\
(4)
\end{array}
\]
\[
\begin{array}{c}
c_1: t_3 := x \\
c_2: y := t_3 + 2
\end{array}
\]

[Location and action labels not required. Note that \( b_1 \) should be considered atomic.]

(b) [Rather than working through the 30 possible interleavings, we observe that the final value may be set by either \( P_1 \) or \( P_3 \). In \( P_3 \), \( x \) may or may not have been incremented before being read resulting in \( y \) being set to the values 2 or 3. If the final value of \( y \) is determined by \( P_1 \), the value of \( y \) may have been read as 0, 2, or 3, before adding 4.]

It is found that the final value of \( y \) may be one of

\[ 2, 3, 4, 6, 7 \]

Question 2.2

(a) \( I \) holds initially since \( x = 0 \land y = 0 \).

Checking all atomic actions:

\( a_1 \): Before the execution, by \( I \) and the condition \( x \leq 2 \) we have \( 0 \leq x \leq 2 \). Incrementing \( x \) by two, we thus get \( 2 \leq x \leq 4 \) after the execution. Since \( y = 2 \) after the execution, \( I \) follows.

\( b_1 \): Since \( 0 \leq y \leq x \) before the execution, setting \( y \) to zero, does not violate \( I \).

\( a_2 \): By the modulo operation, \( 0 \leq x \leq 3 \) after the execution, and since \( y = 0 \), \( I \) holds.

Since \( I \) holds initially and is preserved by all atomic actions, \( I \) is an invariant of the program.

(b) Transition graph:
(c) The predicate \( I \land y \leq 2 \) is seen to encompass all the reachable states and is thus an invariant of the program. However, as it allows states like \((1,1)\), which is not reachable, the predicate is not a characteristic invariant of the program.

(d) Assuming weak fairness

- **F does not hold.** [From the states \((2,2)\) or \((3,2)\) the execution may go to state \((3,0)\) and from there follow the cycle:

\[
(3,0) \xrightarrow{a_2} (0,0) \xrightarrow{a_1} (2,2) \xrightarrow{b_1} (2,0) \xrightarrow{a_2} (3,0) \xrightarrow{a_2} \ldots
\]

Since this execution path does not remain at a single state, it satisfies weak fairness.

Hence, \( y > 0 \) does not necessarily lead to \( x = 1 \).]

- **G holds.** [The only reachable states where the condition \( x = 3 \) can occur are \((3,2)\) and \((3,0)\). From \((3,2)\) weak fairness leads to \((3,0)\) and from \((3,0)\) to \((0,0)\).]

- **H does not hold.** [The infinite execution cycle

\[
(0,0) \xrightarrow{a_2} (1,0) \xrightarrow{a_1} (3,2) \xrightarrow{b_1} (3,0) \xrightarrow{a_2} (0,0) \xrightarrow{a_2} \ldots
\]

satisfies weak fairness but avoids states where \( x = 2 \).]

- **J does not hold.** [In the infinite execution cycle

\[
(0,0) \xrightarrow{a_2} (1,0) \xrightarrow{a_1} (2,0) \xrightarrow{a_2} (3,0) \xrightarrow{a_2} (0,0) \xrightarrow{a_2} \ldots
\]

\( a_1 \) is not continuously enabled and hence the execution satisfies weak fairness while avoiding states where \( y > 0 \).]

Assuming strong fairness

- **F does not hold.** [In the sequence (*) all actions are executed infinitely often and hence strong fairness is satisfied.]

- **G holds.** [By weak fairness.]

- **H does not hold.** [In the sequence (**) all actions are executed infinitely often and hence also strong fairness is satisfied.]

- **J holds.** [The only way to avoid \( y > 0 \) is by the cycle (***)], but in this, \( a_1 \) is infinitely often enabled without being taken. This, however, is not possible assuming strong fairness, and hence \( a_1 \) must be taken over and over again leading to \( \square \Diamond (y > 0) \).]

PROBLEM 3

Question 3.1

(a) Given a global variable

\[
\text{var tilt : integer := 0;}
\]

the operations may be specified by:

\[
\begin{align*}
\text{level():} & \quad \langle \text{tilt} = 0 \rightarrow \text{skip} \rangle \quad \text{[or} \quad \langle \text{await tilt} = 0 \rangle \text{]} \\
\text{left}(k : \text{posinteger}): & \quad \langle \text{tilt} := \text{tilt} + k \rangle \\
\text{right}(k : \text{posinteger}): & \quad \langle \text{tilt} := \text{tilt} - k \rangle
\end{align*}
\]
(b) If \( \text{tilt} \) remains zero, there can be no calls of \( \text{left}() \) or \( \text{right}() \), and hence pending calls of \( \text{level}() \) must be served. Hence the implementation ensures \textit{weak fairness} for the \( \text{level}() \) operation.

On the other hand, if \( \text{left}(1) \) and \( \text{right}(1) \) are called alternatingly, the value of \( \text{tilt} \) will be zero over and over again, but there is not guarantee that pending calls of \( \text{level}() \) are served when this is the case. Therefore, \textit{strong fairness} for \( \text{level}() \) is \textbf{not} guaranteed by the implementation.

(c) In order to empty the \( \text{level}() \) queue of clients when \( \text{tilt} \) becomes zero, the following may be inserted in the server loop:

\[
\text{repeat} \\
\quad \text{if} \; \text{tilt} = 0 \; \text{then} \; \text{for} \; i \; \text{in} \; 1..?\text{level} \; \text{do} \; \text{in} \; \text{level}() \; \rightarrow \; \text{skip} \; \text{ni}; \\
\quad \text{in} \; \text{level}() \; \text{and} \; \ldots \\
\quad \ldots \\
\quad \text{ni} \\
\text{forever}
\]

[The \textit{if} statement may instead be inserted at the end of the loop or within each of the \textit{left} and \textit{right} branches.]

Alternatively, the acceptance of further \( \text{left}() \) or \( \text{right}() \) calls may be suspended while the \( \text{level}() \) queue is served:

\[
\text{repeat} \\
\quad \text{in} \; \text{level}() \; \text{and} \; \text{tilt} = 0 \; \rightarrow \; \text{skip} \\
\quad \left[ \quad \text{left}(k : \text{posinteger}) \; \text{and} \; \neg(\text{tilt} = 0 \; \text{and} \; ?\text{level} > 0) \; \rightarrow \; \text{tilt} := \text{tilt} + k \\
\quad \left] \quad \text{right}(k : \text{posinteger}) \; \text{and} \; \neg(\text{tilt} = 0 \; \text{and} \; ?\text{level} > 0) \; \rightarrow \; \text{tilt} := \text{tilt} - k \\
\quad \text{ni} \\
\text{forever}
\]

\textbf{Question 3.2}

(a) Synchronization code for each process:

\[
P_1: \; \text{Balance.left}(1); \quad P_2: \; \text{Balance.right}(1); \\
\quad \text{Balance.level}(); \quad \text{Balance.level}();
\]

(b) Synchronization code for (say) \( P_1 \) and for the remaining processes:

\[
P_1: \; \text{Balance.left}(n - 1); \quad P_2, P_3, \ldots, P_n: \; \text{Balance.right}(1); \\
\quad \text{Balance.level}(); \quad \text{Balance.level}();
\]

(c) The solution in (b) cannot be used as a normal barrier since a fast process leaving the synchronization point through \( \text{level}() \) may get to the next synchronization point and then "make the balance tilt" before all processes have left the first synchronization point leaving the processes in a kind of deadlock state.

[This race condition can be eliminated by using two distinct one-time barriers in succession at each synchronization point.]
Question 3.3

(a) The Balance module is readily implemented as a monitor:

```
monitor Balance
var tilt : integer := 0;
    Zero : condition;

procedure level() {
    if tilt ≠ 0 then wait(Zero)
}

procedure left(k : posinteger) {
    tilt := tilt + k;
    if tilt = 0 then signal_all(Zero)
}

procedure right(k : posinteger) {
    tilt := tilt - k;
    if tilt = 0 then signal_all(Zero)
}
end
```

[This solution renders the level() operation strongly fair, as all waiting processes are guaranteed to be woken up and leave the monitor when tilt becomes zero.]

(b) Calls of level() should be waiting only if tilt is not zero:

\[ I \triangleq \text{waiting}(Zero) > 0 \Rightarrow tilt ≠ 0 \]

\( I \) holds initially as Zero is empty. The queue size is only incremented when tilt is non-zero and whenever tilt becomes zero, the condition queue is completely emptied. Hence \( I \) is a monitor invariant.

(c) [Normally signal_all may be simulated by repeated signalling until the queue is empty, but not having empty(c) available either, the solution calls for cascaded wake-up. If during the cascade, however, tilt should become non-zero, the cascade must be stopped.]

In the solution given by (a), the level() operation may be replaced by:

```
procedure level() {
    while tilt ≠ 0 do wait(Zero);
    signal(Zero)
}
```

and in the left() and right() operations, signal_all(Zero) should be replaced by signal(Zero).