PROBLEM 1

Question 1.1

A direct translation of the semaphore operations yields:

Recognizing the cross signalling to be equivalent to a true (barrier) synchronization, this may be reduced to:

Question 1.2

The reduced Petri is readily implemented by synchronous communications:

<table>
<thead>
<tr>
<th>process</th>
<th>process</th>
<th>process</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$;</td>
<td>$P_2$;</td>
<td>$P_3$;</td>
</tr>
<tr>
<td>repeat</td>
<td>repeat</td>
<td>repeat</td>
</tr>
<tr>
<td>$A;$</td>
<td>$B;$</td>
<td>$P_2$?()</td>
</tr>
<tr>
<td>$P_2$!()</td>
<td>$P_3$!();</td>
<td>$C;$</td>
</tr>
<tr>
<td>forever</td>
<td>$P_1$?();</td>
<td>$P_2$!()</td>
</tr>
<tr>
<td></td>
<td>$P_3$?()</td>
<td>forever</td>
</tr>
</tbody>
</table>
Question 1.3

The semaphores $SBA$ and $SBC$ cannot be replaced by a common semaphore $SB$, at this would allow for the following scenario:

- $A$ and $B$ are executed concurrently
- $P_B$ executes $V(SC)$ and $V(SA)$
- $P_A$ executes $V(SB)$, passes $P(SA)$, executes $A$ and then $V(SB)$
- $P_B$ may now pass $P(SB)$ twice and then execute $B$

Since $B$ has now been executed twice while $C$ has not been executed at all, $B$ and $C$ no longer alternate.

PROBLEM 2

Question 2.1

(a) Transition diagrams:

\[ P_1: \]

- $a_1: t_1 := x$
- $a_2: t_1 := t_1 + y$
- $a_3: x := t_1 + 1$

\[ P_2: \]

- $b_1: y := y + 3$
- $c_1: t_3 := y$

\[ P_3: \]

- $c_2: x := t_3$

[Left-to-right evaluation assumed. Location and action labels not required. Note that $b_1$ can be considered atomic.]

(b) Rather than working through the 60 (!) possible interleavings, we first observe that $y$ is changed only by $P_2$ and may be read as either 0 or 3 by the other processes. The final value of $x$ is determined by either $a_3$ or $c_2$. In the latter case, $t_3$ may be 0 or 3 and hence $x$ may set to these values. Otherwise, the final value set by $a_3$ is based on the value of $x$ read by $a_1$ and the value of $y$ read by $a_2$. Both may be either 0 or 3 and adding 1, $a_3$ may set $x$ to either 1, 4, or 7.

It is found that the final value of $x$ may be one of

\[ 0, 1, 3, 4, 7 \]
Question 2.2

$P$ is preserved by $a$ and $b$. [By $b$ since $P \land y \leq 0$ imply $x > 0$.]

$Q$ is preserved by $b$ and $c$. [By $b$ since either $x < 0$ is and remains true or otherwise $x \geq 0$ and hence $y \geq 0$ is preserved.]

$R$ is preserved by $a$ only. [Not by $b$ in case $y = 0$.]

Question 2.3

(a) The states $(0,0), (1,1)$ repeated forever will satisfy all parts of $F$.

(b) The strongest invariant corresponds to those conjuncts of $F$ which are of the form $\square H$ where $H$ is a state predicate. Hence the strongest invariant is

$$x \geq 0 \land y \geq x$$

or equivalently

$$0 \leq x \leq y$$

(b) Assuming $F$, the only condition which is guaranteed to remain constantly true is $y > 0 \lor x = 0$. Hence only $a_3$ is guaranteed to be executed under the assumption of weak fairness.

(c) Assuming $F$, $x$ is guaranteed to become 0 over and over again since $x > 0$ implies $y > 0$ which leads to $y = 0$ implying at the same time $x = 0$. Hence $a_1$, $a_3$ and $a_4$ will be eventually executed under the assumption of strong fairness. Nothing, however, forces $x$ ever to become 1.

PROBLEM 3

Question 3.1

(a) The operation $\textbf{op} set(b : \text{posinteger})$; must be added to the interface declarations.

In the outermost $\textbf{in}$ construct, the following branch is added:

$$[] \quad \text{set}(b : \text{posinteger}) \rightarrow k := b$$

[It is not necessary to add the operation to the inner $\textbf{in}$, as this is guaranteed not to block.]

(b) When $k$ is set to 1, the module synchronously transfers a donation from one donor to one donee. Thus, for $k = 1$ $\textbf{Hub}$ is equivalent to a $\text{ synchronous communication channel}$ with $\text{donate}()$ acting as the send operation.
Question 3.2

(a) monitor Hub

```plaintext
var rem : integer := −1; — remaining donors in bundle, −1 if inactive
total : integer := 0;
Donors, DonorsReady, Done : condition;

procedure donate(d : posinteger) {
    if length(Donors) = K_0 − 1 then signal(DonorsReady);
    wait(Donors);
    total := total + d;
    rem := rem − 1;
    if rem = 0 then signal(Done)
}

function receive() returns posinteger {
    while length(Donors) < K_0 do wait(DonorsReady);
    total := 0;
    rem := K_0;
    for i in 1..K_0 do signal(Donors);
    wait(Done);
    rem := −1;
    return total
}
```

[In this solutions donors always wait, but the last donor wakes up the donee which then controls the bundling of donations by releasing the appropriate number of donors before awaiting the total result. The solution is not robust towards spurious wakeups.]

(b) Calls of donate() should be waiting only if the donee is not present or not enough other donors have shown up:

\[ H \triangleq \text{waiting}(\text{Donors}) > 0 \Rightarrow \text{waiting}(\text{DonorsReady}) = 0 \lor \text{waiting}(\text{Donors}) < K_0 \]

or equivalently

\[ H' \triangleq \text{waiting}(\text{Donors}) \geq K_0 \Rightarrow \text{waiting}(\text{DonorsReady}) = 0 \]
(c) If several donees call `receive()` concurrently, they should be served one at a time. Especially, if a donation bundling is taking place, new calls should be held back until the donation has been fully delivered. In the current solution, a bundling is ongoing if and only if \( rem \geq 0 \). Thus, the first line of `receive` could be changed to:

\[
\textbf{while} \ \text{length}(\text{Donors}) < K_0 \lor rem \geq 0 \ \textbf{do} \ \text{wait}(\text{DonorsReady});
\]

Since a new call of `receive()` may now occur during the bundling, one call of `receive` should try to wake the next one. Hence, at the end of `receive()`, a signal should be inserted:

\[
\begin{align*}
\text{rem} & := -1; \\
\text{if} \ \text{length}(\text{Donors}) \geq K_0 \ \text{then} \ \text{signal}(\text{DonorsReady}); \\
\text{return} \ \text{total}
\end{align*}
\]

[Alternatively, further donees may be held in a dedicated pre-queue while a flag signals that one donee is active.]

**Question 3.3**

The `Hub` module effectively synchronizes one donee with \( k \) donors. Hence barrier synchronization may be obtained e.g. by letting \( P_1, \ldots, P_{n-1} \) act as donors and \( P_n \) as donee. Setting \( K_0 \) to \( n - 1 \), the barrier synchronization code then becomes:

\[
\begin{align*}
\text{process} \ P_i[i : 1..n-1] & \quad \text{process} \ P_n \\
\text{Hub}.\text{donate}(1); & \quad x := \text{Hub}.\text{receive}() \\
\text{Hub}.\text{receive}(); & \quad \text{Hub}.\text{donate}(1);
\end{align*}
\]

[Here \( x \) is a dummy variable and the donation is set to an arbitrary positive number.]