PROBLEM 1

Question 1.1

type SigType = A | B;

monitor SigBuf

var a, b : integer := 0; — signal counters

RoomA, RoomB, SigReady : condition;

procedure put(sig : SigType) {

while a + b ≥ M do if sig = A then wait(RoomA) else wait(RoomB);

if sig = A then a := a + 1
else b := b + 1;

signal(SigReady)

}

function get() returns SigType {

var res : SigType;

while (a = 0 ∧ b = 0) do wait(SigReady);

if a > 0 then a := a - 1; res := A
else b := b - 1; res := B;

if nonempty(RoomA) then signal(RoomA)
else signal(RoomB);

return res

}

end

Question 1.2

(a) \[ I \triangleq a \geq 0 \land b \geq 0 \land a + b \leq M \]

I holds initially since \( a = 0 \land b = 0 \). Since a is decremented only if \( a > 0 \), \( a \geq 0 \) is satisfied. \( b \) is decremented when \( \neg(a = 0 \land b = 0) \) and \( \neg(a > 0) \), which together with I implies \( b > 0 \). Hence \( b \geq 0 \) is maintained. Finally \( a \) or \( b \) is incremented only when \( a + b < M \) and hence \( a + b \leq M \) is preserved.

(b) Calls of get() should wait only if there are no signals of any kind. This can be expressed by:

\[ J \triangleq waiting(SigReady) > 0 \Rightarrow a + b = 0 \]

However, since the signals are removed only after the wait, the invariant should be weakened:

\[ \hat{J} \triangleq waiting(SigReady) > 0 \Rightarrow a + b = 0 \lor woken(SigReady) > 0 \]
**Question 1.3**

The following procedure is added to the monitor:

```
procedure dropB() {
    var k : int = b;
    b := 0;
    while k > 0 ∧ nonempty(RoomA) do {signal(RoomA); k := k - 1};
    if k > 0 then signal_all(RoomB)
}
```

[Here, the number of released signal slots \(b\) is recorded and as many as possible of these given to processes waiting for room for \(A\)-signals. Any remaining slots are given to \(B\)-signals. Alternatively, this effect may be achieved with cascaded wakeup in \(put\).]

**PROBLEM 2**

**Question 2.1**

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

All three \(a\)-actions are potentially dangerous for \(I\):

\(a_1\): Is executed only if \(y = 0\) and does not change \(y\). Thus \(y = 0\) after the action and \(I\) holds.

\(a_2\): If \(x = 0\) when executed, \(y = 0\) after and \(I\) holds. If \(x > 0\) when executed, \(y > 0 ∧ x = 0\) after, i.e. after the action \(x ≠ y\) and \(I\) holds

\(a_3\): After this action we always have \(x = 1 ∧ y = 2\), thus \(I\) holds after the actions.

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

(b) Transition diagram:

```
\(a_2\)
(0, 2) \rightleftharpoons (1, 2)
\(a_2\)
(0, 1) \rightleftharpoons (1, 0)
\(a_2\)
(0, 0) \rightleftharpoons (2, 0)
\(a_1\)
```

The initial state is \((0, 0)\). Further there are an \(a_2\) self-loop on the state \((0, 0)\) and an \(a_3\) self-loop on state \((1, 2)\) (not shown).

(c) From the transition graph, we see that the state \((x, y) = (1, 2)\) is reachable and therefore \((x = 0 ∨ y = 0)\) is not an invariant of the program.
Question 2.2

(a) Given a transition graph, weak fairness ensures that the execution cannot remain forever in a state where there are enabled actions leading to other states. By inspecting the possible execution paths in the transition graph we therefore conclude that any execution must pass through \((x, y) = (0, 0)\) over and over again. Thus, \(\square \Diamond y = 0\) is a property of the program.

(b) Likewise, any execution will have to pass through \((x, y) = (1, 0)\) over and over again. Therefore \(a_3\) will be enabled infinitely often and by strong fairness we then get that \(a_3\) must be taken infinitely often. We therefore get to \((x, y) = (1, 2)\) infinitely often, i.e. \(\square \Diamond y = 2\) holds for the program.

Question 2.3

(a) \(I\) is violated by the interleaving:

\[
\begin{align*}
(0, 0) & \xrightarrow{b_1} (0, 0) \xrightarrow{c_1} (0, 0) \xrightarrow{d_1} (1, 0) \xrightarrow{b_1} (1, 0) \xrightarrow{c_1} (1, 0) \xrightarrow{a_2} (0, 1) \xrightarrow{d_1} (1, 1)
\end{align*}
\]

(b) \(H\) can be defined as:

\[
H \triangleq (x \leq 2) \land (at \ c_1 \lor at \ d_1 \Rightarrow x < 2)
\]

PROBLEM 3

Question 3.1
Question 3.2

Corresponding to the Petri-net, we introduce a semaphore $SB$ that counts the number of $A$-operations executed. $B$ may then be executed after $\text{maj}(n)$ $P$-operations on $SB$. $Q$ controls the final synchronization by awaiting the remaining $A$-operations and then signalling each $P$ process.

\begin{verbatim}
var $SB$ : semaphore; Counts no. of $A$ done
$SA[1..n]$ : semaphore; OK to start $A_i$ again.

All semaphores are initialized to 0

process $P[i : 1..n] =$
repeat
$A_i;$
$V(SB);$ P($SA[i]$)
forever;

process $Q =$
repeat
for $j$ in $1..\text{maj}(n)$ do $P(SB);$ B;
for $j$ in $(\text{maj}(n) + 1)..n$ do $P(SB);$ for $j$ in $1..n$ do $V(SA[j])$
forever;

[It is not possible to replace $SA[1..n]$ with a common semaphore since a $P$ process may wait again immediately after a wait and thereby could consume a token destined for another process.]
\end{verbatim}

Question 3.3

\begin{verbatim}
module Synch
  op $Apause() ;$
op Bstart();
  op Bend();
body
  process $Server =$
repeat
  in Bstart() and ?$Apause \geq \text{maj}(n)$ \rightarrow skip ni;
in Bend() and ?$Apause = n$ \rightarrow skip ni;
for $j$ in $1..n$ do
  in $Apause() \rightarrow skip ni
forever;
end Synch;
\end{verbatim}
PROBLEM 4

Question 4.1

(a) If the manager filled the bag via the \textit{Put} operation, we could risk that none of the workers would be able to return a result. This would block the computation.

(b) The manager can be allowed to fill the bag, if there are idle workers since these are known to call \textit{GetPair}. The synchronization expression would be:

\[ |bag| < K - 1 \lor (|bag| = K - 1 \land working < M) \]

(c) If the workers took the elements one at a time, and if there were only two elements left, two workers could take one each leading to a deadlock.

Question 4.2

(a) A new branch is added to the server’s \texttt{in}-statement:

\[
\begin{array}{l}
[] \ TryGetElem(\texttt{var e}) \textbf{returns ok} \rightarrow \textbf{if} |bag| > 0 \textbf{ then } e := \text{remove}(bag); \text{ }
\text{ok := true } \text{ }
\text{else } \text{ok := false }
\end{array}
\]

(b) \texttt{process Worker[1..M]} =

\[
\begin{array}{l}
\text{var a, b, c : integer;}
\text{repeat}
\text{Control.GetPair(a, b);}
\text{c := a \oplus b;}
\text{while Control.TryGetElem(a) do c := c \oplus a;}
\text{Control.Result(c)}
\text{forever;}
\end{array}
\]

Question 4.3

(a) In \textit{Put}:

\[
\text{if } |bag| \geq 2 \textbf{ then signal(PairReady)}
\]

In \textit{GetResult}:

\textbf{No signalling needed (computation over)}

In \textit{GetPair}:

\[
\begin{array}{l}
\text{if empty(Room) then signal(PutOK)}
\text{else } \{ \text{signal(Room); signal(Room) } \}
\end{array}
\]

In \textit{Result}:

\[
\begin{array}{l}
\text{if } |bag| \geq 2 \textbf{ then signal(PairReady)};
\text{else if working = 0 then signal(ResultOK)}
\end{array}
\]

\textbf{Comments:} In this solution, any waiting condition that may have become true by an operation is signalled. In the \textit{GetPair} case, both \textit{Put} and \textit{Result} may benefit from the
two new free slots in the bag. Here it has been decided to give priority to calls of *Result* to keep the workers busy. A subtle point is that if the *Room* queue is not empty, the bag was previously full (or would be filled by awakened calls of *Result*, see *I* below). Even if there is only one call waiting in *Room*, at most one free slot will be left in the bag, and therefore there is no need to consider *PutOK* in this case.

The manager will signal *PairReady* whenever an element is put in the bag once $|\text{bag}| \geq 2$ although only every second element gives a pair. This may lead to unnecessary wake-ups if the manager inserts elements faster than they are taken out. Such a situation, however, may occur only in the starting phase when the workers are still idle and thus need not be considered a significant inefficiency.

(b) A call of *Result* would wait unnecessarily if there was room in the bag. However this is to hold only when all woken calls of *Result* have reentered the monitor. This can be expressed by:

$$I \triangleq \text{waiting}(\text{Room}) > 0 \Rightarrow |\text{bag}| = K \lor \text{woken}(\text{Room}) > 0$$

**Remark:** Although $I$ is an invariant for the monitor, it is not strong enough to be proven by an inductive, operational argument. For this, the invariant must express that there are enough woken calls to fill the bag, e.g.

$$I' \triangleq \text{waiting}(\text{Room}) > 0 \Rightarrow |\text{bag}| + \text{woken}(\text{Room}) \geq K$$

This invariant can be proven to be preserved by all operations.