PROBLEM 1 (approx. 15 %)

In a system, a number of processes generate signals of two kinds, A-signals and B-signals. The generated signals should be buffered by a component $\text{SigBuf}$ and from there retrieved by other processes of the system for further processing. The $\text{SigBuf}$ component thus has two operations: $\text{put}(\text{sig})$ which adds a signal $\text{sig}$ to the buffer and $\text{get}()$ which retrieves a buffered signal. The buffer has a limited capacity of $M$ ($M > 0$) signals. Here, signals are just represented by two enumeration constants, A and B. Signals of kind A are considered most important and should be given preference.

The behaviour of the signal buffer is specified as shared object with conditional atomic actions:

```
type $\text{SigType} = \text{A} | \text{B};

object $\text{SigBuf}$;
    var $a, b : integer := 0; \quad \text{— signal counters}
    op $\text{put}(\text{sig} : \text{SigType})$
    \{ a + b < M \rightarrow \text{if } \text{sig} = \text{A} \text{ then } a := a + 1 \\
        \text{else } b := b + 1\}
    op $\text{get}()$ returns $\text{SigType}$
    \{ a > 0 \lor b > 0 \rightarrow \text{if } a > 0 \text{ then } a := a - 1; \quad \text{return } \text{A} \quad \text{— prefer kind A} \\
        \text{else } b := b - 1; \quad \text{return } \text{B}\}
end
```

Question 1.1:

Implement the $\text{SigBuf}$ as a monitor according to the specification. Where the specification may leave you with a choice, handling of A-signals should be given priority.

Question 1.2:

(a) State a predicate $I$ which characterizes the possible combinations of $a$ and $b$.

Argue that $I$ is an invariant of the monitor.

(b) State a monitor invariant which expresses that calls of $\text{get}()$ do not wait unnecessarily.

Question 1.3:

Show how to extend your monitor solution with a procedure $\text{dropB}()$ which discards all buffered B-signals (with the intention of making room for more A-signals).
PROBLEM 2 (approx. 20 %)

Consider the concurrent program:

```plaintext
var x, y : integer := 0;

co
  repeat a1: ⟨x < 2 ∧ y = 0 → x := x + 1⟩ forever
  ∥
  repeat a2: ⟨y := x; x := 0⟩ forever
  ∥
  repeat a3: ⟨x = 1 → y := 2⟩ forever
oc
```

Question 2.1:

(a) Prove inductively that $I \triangleq (y = 0 ∨ x \neq y)$ is an invariant of the program.

(b) Draw the (reachable part of) the transition graph for the program. Since control remains at the $a$-actions, only the $(x, y)$ part of the state needs to be shown.

(c) Determine from the transition graph, whether $(x = 0 ∨ y = 0)$ is an invariant of the program.

Question 2.2:

(a) Argue that $\Box蟾 y = 0$ holds for the program under the assumption of weak fairness.

(b) Determine whether $\Box蟾 y = 2$ holds for the program under the assumption of strong fairness.

Question 2.3:

Assume that the program is modified by replacing the action $a_1$ by the refinement:

```plaintext
b_1: ⟨await x < 2⟩;  c_1: ⟨await y = 0⟩;  d_1: ⟨x := x + 1⟩
```

(a) Show that $I$ is not an invariant of the modified program.

(b) We would like to prove that $x \leq 2$ is an invariant of the modified program.

State a predicate $H$ that (i) implies $x \leq 2$, (ii) holds initially, and (iii) is inductive for the modified program.

[For $H$ to be inductive, it must be strong enough to be preserved by all atomic actions, but you need not demonstrate this.]
PROBLEM 3 (approx. 30 %)

In a system, a number of operations $A_1, A_2, \ldots, A_n$ ($n \geq 1$) plus an operation $B$ are to be executed the following way:

(*) $A_1, A_2, \ldots, A_n$ are executed concurrently. As soon as a majority of these have finished, $B$ can be executed concurrently with the remaining $A$-operations. When all the operations $A_1, A_2, \ldots, A_n$, and $B$ have finished, the execution starts all over again.

As usual, majority is given by the number $\text{maj}(n) \triangleq n/2 + 1$, where / is integer division.

Question 3.1:

For a system with $n = 3$, draw a Petri Net in which the four operations $A_1, A_2, A_3$, and $B$ are synchronized as described by (*). In the net, the operations should be represented by transitions.

Question 3.2:

The operations are to be executed by $n$ sequential processes $P_1, P_2, \ldots, P_n$ plus a sequential process $Q$. The form of these processes are:

$$
\text{process } P[i : 1..n] = \\
\text{repeat } A_i \\
\text{forever}; \\
\text{process } Q = \\
\text{repeat } B \\
\text{forever};
$$

Show how to synchronize these processes using semaphores so that the operations $A_i$ ($i : 1..n$) and $B$ become synchronized as described by (*).

Question 3.3:

The processes $P_1, P_2, \ldots, P_n$ and $Q$ are now to be synchronized using a module $\text{Synch}$ instead. The module has three parameterless operations $\text{Apause}()$, $\text{Bstart}()$, and $\text{Bend}()$ to be called by the processes as shown below:

$$
\text{module } \text{Synch} \\
\text{op } \text{Apause}(); \\
\text{op } \text{Bstart}(); \\
\text{op } \text{Bend}(); \\
\text{end};
$$

$$
\text{process } P[i : 1..n] = \\
\text{repeat } A_i; \\
\text{Synch.Apause()} \\
\text{forever}; \\
\text{process } Q = \\
\text{repeat } \text{Synch.Bstart}(); \\
\text{B}; \\
\text{Synch.Bend()} \\
\text{forever};
$$

Write a server process for the module $\text{Synch}$ so that the operations $A_i$ ($i : 1..n$) and $B$ become synchronized as described by (*).

[You may assume that synchronization expressions in rendezvous guards that use the expression $?op$ for the number of pending invocations of $op$ are reevaluated whenever this number changes.]
PROBLEM 4  (approx. 35 %)

An expression over integer elements of the form

\[ a_1 \oplus a_2 \oplus \ldots \oplus a_n \]

can be computed by a variant of the bag-of-tasks model if the \( \oplus \) operator is associative and commutative so that it does not matter in which order the elements are combined. Examples of such operators are addition and maximum.

Below, such computations are carried out by \( M \ (M \geq 1) \) worker processes and a manager process interacting through a bag of integers maintained by a server module.

```plaintext
process Manager =
  var A[1..n], r : integer;
repeat
  read A[1..n];
  for i in 1..n do Control.Put(A[i]);
  Control.GetResult(r);
  print r
forever;

process Worker[1..M] =
  var a, b, c : integer;
repeat
  Control.GetPair(a, b);
  c := a \oplus b;
  Control.Result(c)
forever;

module Control
  op Put(integer);
  op GetResult(var integer);
  op GetPair(var integer, var integer);
  op Result(integer);

body

process Server =
  var bag : Bag[K];
  working : integer := 0;
repeat
  in Put(e) and |bag| < K – 1 \rightarrow insert(bag, e)
  \[ GetResult(var r) and |bag| = 1 \land working = 0 \rightarrow r := remove(bag) \]
  \[ GetPair(var a, var b) and |bag| \geq 2 \rightarrow a := remove(bag);
      b := remove(bag);
      working := working + 1 \]
  \[ Result(c) and |bag| < K \rightarrow insert(bag, c);
      working := working – 1 \]

end Control;
```

Here \( Bag[K] \) is a data type of integer bags/multi-sets with capacity \( K \ (K \geq 3) \). The type has the standard operations \( insert(bag, e) \) that inserts the integer element \( e \) into \( bag \), and \( remove(bag) \) that returns an element of \( bag \) and removes it. Further, \( |bag| \) gives the current number of elements in the bag. Initially, the bag is empty. It is an error to remove from an empty bag or to insert into a full bag.

The problem is continued on the next page
Question 4.1:

(a) Explain the reason why the synchronization expression for \textit{Put} does not allow the manager to fill the bag.

(b) State a weaker synchronization expression for \textit{Put} that will allow the manager to fill the bag under certain conditions.

(c) Why do the workers acquire the two elements as a pair rather than one at a time?

Question 4.2:

In the given system, a worker returns a result and then immediately asks for a new pair of elements. This can be improved upon by letting a worker combine more elements before returning a result. To enable this, the module \textit{Control} is to be extended with an operation

\begin{verbatim}
   op TryGetElem(var integer) returns boolean
\end{verbatim}

where \textit{TryGetElem(e)} should return true if it is immediately possible to take a single element from the bag (returned in \(e\)) and false otherwise (in which case \(e\) is unchanged).

(a) Describe how to modify the server process to handle \textit{TryGetElem}. The functionality of the given operations must remain unchanged.

(b) Given the extended module, show the protocol to be used by the workers in order to combine as many elements as possible before returning a result.

\textit{The problem is continued on the next page}
**Question 4.3:**

The given module *Control* is now to be replaced by a monitor. A partial implementation of such a monitor is shown below:

```
monitor Control

var bag : Bag[K];
    working : integer := 0;
    PutOK, ResultOK, PairReady, Room : condition;

procedure Put(e : integer) {
    while |bag| ≥ K − 1 do wait(PutOK);
    insert(bag, e);
    ...
}

procedure GetResult(var r : integer) {
    while |bag| ≠ 1 ∨ working ≠ 0 do wait(ResultOK);
    r := remove(bag);
    ...
}

procedure GetPair(var a : integer, var b : integer) {
    while |bag| < 2 do wait(PairReady);
    a := remove(bag);
    b := remove(bag);
    working := working + 1;
    ...
}

procedure Result(c : integer) {
    while |bag| ≥ K do wait(Room);
    insert(bag, c);
    working := working − 1;
    ...
}
end
```

(a) Finish the implementation of the monitor *Control* by stating for each procedure the missing signalling code (indicated by dots) so that the monitor corresponds to the module *Control*. The monitor should use signal-and-continue semantics and unnecessary wake-ups should be avoided. Signalling on empty queues is not considered harmful.

(b) State a desired monitor invariant $I$ expressing that calls of *Result* do not wait unnecessarily.