1. (a) 6 marks A binary semaphore \texttt{BinSem} can be used for synchronization and mutual exclusion. Using pseudo code, sketch out the pattern for synchronization and for mutual exclusion using \texttt{BinSem}

(b) 1 mark Explain the difference between a \textit{simple} and \textit{complex} critical section.

(c) 7 marks The following precedence graph shows the optimal concurrency possible for a series of statements \texttt{S1..S7}:

\begin{center}
\begin{tikzpicture}
\node (S1) at (0,0) {$S1$};
\node (S2) at (1,0) {$S2$};
\node (S3) at (1,-1) {$S3$};
\node (S4) at (0,-2) {$S4$};
\node (S5) at (2,-2) {$S5$};
\node (S6) at (0,-3) {$S6$};
\node (S7) at (2,-3) {$S7$};
\draw[dashed] (S1) -- (S3);
\draw[dashed] (S2) -- (S3);
\draw (S3) -- (S4);
\draw (S3) -- (S5);
\draw (S5) -- (S6);
\draw (S5) -- (S7);
\end{tikzpicture}
\end{center}

Code the statements using only \texttt{COBEGIN} and \texttt{COEND} in conjunction with \textit{binary} semaphores using \texttt{P} and \texttt{V} to achieve the concurrency of the precedence graph. Use pseudo-code for this problem, not \texttt{µC++}. Use \texttt{BEGIN} and \texttt{END} to make several statements into a single statement and show the initial value (0/1) for all semaphores. Name your semaphores \texttt{Ln}, e.g., \texttt{L1,L2,...}, to simplify marking.

2. (a) 2 marks Briefly explain the terms \textit{split-binary semaphore} and \textit{baton passing}.

(b) 1 mark Explain the \texttt{µC++} mechanism to easily implement a \textit{shadow queue}.

(c) 2 marks To prevent staleness/freshness, some algorithms have a special condition variable that has only 0 or 1 tasks waiting on it (sometimes called a chair). Explain the purpose of this condition variable.

(d) 2 marks On August 14, 2003, the Northeastern blackout occurred, which was the worst power outage in North American history. What concurrent error occurred that caused the blackout, and why is this error so difficult to debug/prevent?

(e) 4 marks Given this 4-way stop, where the cars are making \textit{no progress}:

\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {$\bigcirc$};
\node (B) at (1,0) {\texttt{[} \texttt{]} \texttt{]}};
\node (C) at (2,0) {$\bigcirc$};
\node (D) at (3,0) {\texttt{[} \texttt{]} \texttt{]}};
\node (E) at (0,-1) {$\bigcirc$};
\node (F) at (1,-1) {\texttt{[} \texttt{]} \texttt{]}};
\node (G) at (2,-1) {$\bigcirc$};
\node (H) at (3,-1) {\texttt{[} \texttt{]} \texttt{]}};
\end{tikzpicture}
\end{center}

i. What must have occurred to start this scenario?

ii. What traffic law (rule) results in no progress after the initial condition?

iii. What is the concurrency name for this scenario?

iv. Suggest a way to prevent this scenario.
(f) 2 marks Briefly explain the difference between *deadlock prevention* and *deadlock avoidance*.

(g) 4 marks Does the following resource allocation graph produce a deadlock? Explain your answer by using graph reduction.

![Resource Allocation Graph](image)

3. (a) 2 marks Explain the difference between *daisy-chain* and *multiple* signalling.

(b) 2 marks What type of problem *cannot* be solved by using *external scheduling*?

(c) 2 marks How does a µC++ monitor prevent *barging*?

(d) When implementing the *implicit (automatic)-signal* monitor using internal scheduling, all the internal tasks must be restarted on a WAITUNTIL and RETURN.
   i. 2 marks Is the restarting busy waiting? Explain.
   ii. 1 mark What is the hardest part of this cyclic restart operation?

(e) 8 marks Using µC++, write the shortest possible *implicit (automatic)-signal* monitor using the following interface to implement a counting semaphore (you may add code anywhere).

```cpp
_Monitor semaphore {
    public:
        semaphore( int cnt = 1 ) { ... } // WRITE THESE ROUTINES
        void P() { ... }
        void V() { ... }
};
```

Assume the existence of the following preprocessor macros for implicit (automatic) signalling:

```cpp
#define AUTOMATIC_SIGNAL ... 
#define WAITUNTIL( cond ) ... 
#define RETURN( expr... ) ... // gcc variable number of parameters
```

Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to delay until the cond evaluates to true. Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value.

4. (a) 4 marks Given the following accept statement:

```cpp
for ( ;; ) {
    _Accept( A ) { ... } 
    or _Accept( B ) { ... }
    _Else { ... }
}
```

i. Is there any implied priority, with respect to accepting calls?

ii. Is there a way to recode this to make it fair?

iii. If the or is removed, explain the behaviour of the statements.

iv. When should the _Else clause be used?
(b) Given the outline for an administrator task:

```cpp
(Task Admin {
    uCondition bench; // workers wait here
    public:
        Job work( . . . ) { // called by worker task
            bench.wait(); // wait for administrator
            return job; // return new work
        }
    private:
        void main() {
            for ( ;; ) {
                ... _Accept( work ); ... // restart worker
                if ( . . . ) bench.signal();
                ... // ... etc
            }
        }
};
```

i. 2 marks The given code does not work properly. Explain why.

ii. 2 marks Correct the line of code so it works properly and explain why this correction fixes the problem.

(c) 3 marks

i. Explain why an asynchronous call increases concurrency?

ii. What is the general mechanism for implementing an asynchronous call?

iii. What is the most complex part of an asynchronous call?

(d) 1 mark Once a future has a value, why it is immutable?

(e) 2 marks Explain the usage difference between the \texttt{\_Accept} and \texttt{\_Select} statements.

5. (a) 2 marks Give two examples why preventing false sharing is difficult.

(b) 2 marks What is a memory model?

(c) 2 marks The TSO memory-model allows reads (\texttt{R}) to be move before disjoint writes (\texttt{W}). Explain how this optimization causes the following code at the start of Dekker’s algorithm to fail.

```cpp
me = WantIn; // W
while ( you == WantIn ) { // R
```

(d) 2 marks For computers with few registers, why are practically all variables implicitly volatile?

(e) 1 mark C++11, Java, Pthreads all have barging. Why?

(f) 1 mark State the main hardware architectural difference between CPUs and GPUs that makes GPUs difficult to program.

6. 22 marks At West Mulubarose, there is a one-lane covered-bridge. When cars arrive at the bridge, drivers can see through the bridge to determine if there are cars on the bridge and which direction the cars are moving, i.e., toward or away from the driver. Drivers must ensure mutual exclusion with respect to direction to ensure there are no head-on collisions. As well, drivers must ensure fair usage of the bridge, so after \( N \) cars have crossed the bridge in one direction the direction must switch, if there are waiting cars on the other side of the bridge.

Simulate the bridge critical-section using a monitor with the following public interface (you may add only a public destructor and private members):
# Monitor Bridge {
    const unsigned int N; // cars before switching direction
    ...  // YOU ADD DECLARATIONS/MEMBERS

    public:
    Bridge( int N ) : N( N ) {}  
    _Nomutex void cross( bool direction );  // YOU WRITE THIS MEMBER

};

A car calls member cross with the desired direction of travel (true or false) to cross the bridge. When cross returns, the car has crossed the bridge in the appropriate direction. Internally, member cross must block the invoking car task until it is appropriate for it to cross the bridge. As a car crosses the bridge, execution the following:

    uThisTask().yield( rand() % 10 ); // pretend to cross bridge

Your solution must NOT have starvation, staleness or freshness, i.e., it MUST service waiting cars in temporal order only on each side of the bridge. Hint, use two counters: one to know how many cars have started crossing the bridge and one to know how many cars have finished crossing.

Implement the bridge monitor in µC++ using internal scheduling to schedule the crossing of the cars using. Write only the Bridge monitor, do not write the car task or program main.

7. 26 marks Write an administrator task for the Couples-Are-Too dating service. At Couples-Are-Too, girls and boys are matched using an integer compatibility code, between 0-4 inclusive, based on a scientific analysis of a personal questionnaire. Clients are compatible if their compatibility codes are equal. To ensure safety and decorum, each couple with matching compatibility code is accompanied on a date by a chaperone. Once all three people are available for a date, the location of the date is returned to each person; the location is an integer value starting at 1 and increasing by 1 for each new date arranged by the dating service.

Figure 1 contains the starting code for the dating-service administrator task (you may not change this interface; you may add private members in the task). (Do not copy the starting code into your exam booklet.) When a client (girl/boy) wants to go on a date, they call the date member of the administrator task with the client’s sex and compatibility code. A future location is returned immediately to the client, so the client can execute asynchronously (e.g., get ready for the date) before accessing the location of the date. That future is inserted into the people matrix, which is lists of unpaired requests. After an insertion, if the administrator detects a matching pair of requests (equal compatibility codes), both future requests are moved to the pairs list. When the administrator detects there is a compatible client waiting for a date and a chaperone is available, a couple is removed from the pairs list and the location of the date is inserted into both futures for the clients and returned directly to the chaperone. After a chaperone has finished supervising a date, it calls the chaperone member to start a new date. The chaperones wait until there is an available couple and the administrator returns the actual location of a date rather than a future date because the chaperone has nothing else to do between dates. In general, there are fewer chaperones than clients so clients may have to wait when they access their future location. Note, there are a couple of places with common code, so think about helper routines.

When the administrator’s destructor is called, the administrator may assume no new calls will occur from chaperones, girls or boys. However, the administrator must wake up any blocked chaperones and return -1 as the date location, and for blocked clients, return a future location containing the exception Closed, indicating they were unsuccessful in obtaining a date.

Ensure the administrator task does as much administration works as possible; a monitor-style solution will receive little or no marks. Write the code for CouplesAreToo::main and any necessary
_Task CouplesAreToo {
  public:
    enum { NoOfCCodes = 5 }; // => 5 compatibility codes, between 0–4
    typedef Future_ISM<int> Flocn; // future location
    _Event Closed {}; // indicate dating-service closed
  private:
    list<Flocn> people[2][NoOfCCodes]; // unpaired client futures
    list<Flocn> pairs; // paired client futures
    uCondition chapers; // bench for chaperones
    int locn; // location of date
    bool sex; // communication variables
    unsigned int ccode;
  public:
    Flocn date( bool sex, unsigned int ccode ) { // called by girls/boys
      CouplesAreToo::sex = sex;
      CouplesAreToo::ccode = ccode;
      Flocn fl;
      people[sex][ccode].push_back( fl ); // store future
      return fl; // return future location of date
    }
    int chaperone() { // called by chaperone
      chapers.wait(); // wait to be processed by administrator
      return locn; // return location of date
    }
  private:
    // YOU MAY ADD OTHER DATA AND Routines
    void main(); // YOU WRITE THIS ROUTINE
};

Figure 1: Couples-Are-Too Administrator

declarations-initializations, **do not write a girl, boy, chaperone task or program main.** Assume the
program main creates and deletes all the necessary tasks, appropriately.

µC++ future server operations are:

- delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients
  waiting for the result.
- exception( uBaseEvent *cause ) – copy a server-generated exception into the future, and the
  exception cause is thrown at clients accessing the future.

C++ list operations are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size()</td>
<td>list size</td>
</tr>
<tr>
<td>bool empty()</td>
<td>size() == 0</td>
</tr>
<tr>
<td>T front()</td>
<td>first element</td>
</tr>
<tr>
<td>T back()</td>
<td>last element</td>
</tr>
<tr>
<td>void push_front( const T &amp;x )</td>
<td>add x before first element</td>
</tr>
<tr>
<td>void push_back( const T &amp;x )</td>
<td>add x after last element</td>
</tr>
<tr>
<td>void pop_front()</td>
<td>remove first element</td>
</tr>
<tr>
<td>void pop_back()</td>
<td>remove last element</td>
</tr>
<tr>
<td>void clear()</td>
<td>erase all elements</td>
</tr>
</tbody>
</table>