```c
monitor dining_controller;
cond ForkReady[5];
boolean fork[5] = {true};
void get_forks(int pid)
{
    int left = pid;
    int right = (pid++) % 5;
    /* grant the left fork */
    if (!fork(left)
        cwait(ForkReady[left]);
    fork(left) = false;
    /* grant the right fork */
    if (!fork(right)
        cwait(ForkReady[right]);
    fork(right) = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (pid++) % 5;
    /* release the left fork */
    if (empty(ForkReady[left])
        fork(left) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /* release the right fork */
    if (empty(ForkReady[right])
        fork(right) = true;
    else
        csignal(ForkReady[right]);
}

void philosopher[k=0 to 4]
{
    while (true)
    {
        <think>;
        get_forks(k);
        <eat spaghetti>;
        release_forks(k);
    }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor
TEST-and-SET

A process would test the condition code using the TS instruction before entering a critical region.

Drawbacks: 1) when many processes are waiting to enter a critical region, starvation could occur (unless FCFS policy is enforced).

2) Waiting processes remain in unproductive, resource-consuming wait loops — busy wait.

WAIT-and-SIGNAL

Two new operations, which are mutually exclusive, are introduced: WAIT and SIGNAL.

WAIT is activated when the process encounters a busy condition code.

SIGNAL is activated when a process exits the critical region and the condition code is set to 'free'.

The whole procedure is finished by Process Scheduler.
Semaphore

The semaphore used by railroads indicates whether the train can proceed. If it is raised the train can continue, but when it's lowered an oncoming train is expected.

(a) Stop  (b) All Clear

Dijkstra's P, V operations:

S — a semaphore variable.

V(s) : S ← S + 1

P(s) : If S > 0 then S ← S - 1
      If S = 0 then Wait

Traditionally, P, V operations are used to enforce mutual exclusion. So S is usually called mutex.

<table>
<thead>
<tr>
<th>State number</th>
<th>Calling process</th>
<th>Operation</th>
<th>Running in critical region</th>
<th>Blocked on s</th>
<th>Value of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1</td>
<td>P (s)</td>
<td>P1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>P1</td>
<td>V (s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P1</td>
<td>V (s)</td>
<td>P2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>P2</td>
<td>P (s)</td>
<td>P2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>P3</td>
<td>P (s)</td>
<td>P2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>P4</td>
<td>P (s)</td>
<td>P3, P4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>P2</td>
<td>V (s)</td>
<td>P3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>P3</td>
<td>V (s)</td>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>P3</td>
<td>V (s)</td>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>P4</td>
<td>V (s)</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Producers and Consumers

The task can be implemented using 2 semaphores:
1. Full — number of full positions in the buffer
2. Empty — number of empty positions in the buffer
The 3rd semaphore will ensure mutual exclusion.
3. Mutex

Here are the definitions of the producer and consumer processes:

**PRODUCER**
produce data
P (empty)
P (mutex)
write data into buffer
V (mutex)
V (full)

**CONSUMER**
P (full)
P (mutex)
read data from buffer
V (mutex)
V (empty)
consume data

Here are the definitions of the variables and functions used in the following algorithm:

Given: Full, Empty, Mutex defined as semaphores
n: maximum number of positions in the buffer
V (x): \( x = x + 1 \) (x is any variable defined as a semaphore)
P (x): if \( x > 0 \) then \( x = x - 1 \)

**COBEGIN** and **COEND** are delimiters used to indicate sections of code to be done concurrently
mutex = 1 means the process is allowed to enter critical region
And here is the algorithm that implements the interaction between producer and consumer:

```plaintext
empty := n
full := 0
mutex := 1
COBEGIN
    repeat until no more data PRODUCER
    repeat until_buffer is empty CONSUMER
COEND
```

**Example**

\[ n = 3 \]

- empty = 3
- full = 0
- mutex = 1

\[
\begin{cases}
\text{PRODUCER: } V(full) : \text{full} \leftarrow 1 \\
\quad \text{//produce data}
\end{cases}
\]

\[
\begin{cases}
\text{Consumer: } P(full) : \text{full} \leftarrow 0 \\
\quad \text{//consume data}
\end{cases}
\]

\[
\begin{cases}
\text{Consumer: } P(full) : \text{Wait} \\
\quad \text{//consumer wants to consume data, but has to wait just as there is nothing available}
\end{cases}
\]
Readers and Writers

Example: Airline reservation system — many readers, a few writers.

Solution 1: Readers are kept waiting only if a writer is modifying the data. Problem? Writer Starvation

Solution 2: Once a writer arrives, readers that are active are allowed to finish processing, but all additional readers are put on hold. Problem? Reader Starvation

Solution 3: When a writer is finished, all readers who are waiting, or "on hold," are allowed to read. When that group of readers is finished, the writer who is "on hold" can begin, and any new readers must wait until the writer is finished.

The state of the system can be summarized by 4 counters initialized to 0.

1. Number of readers who have requested a resource and haven’t yet released it (R1=0);
2. Number of readers who are using a resource and haven’t yet released it (R2=0);
3. Number of writers who have requested a resource and haven’t yet released it (W1=0);
4. Number of writers who are using a resource and haven’t yet released it (W2=0).