Here is an example of a situation where deadlock can occur.

Mutex M1, M2;

/* Thread 1 */
while (1) {
    NonCriticalSection()
    Mutex_lock(&M1);
    Mutex_lock(&M2);
    CriticalSection();
    Mutex_unlock(&M2);
    Mutex_unlock(&M1);
}

/* Thread 2 */
while (1) {
    NonCriticalSection()
    Mutex_lock(&M2);
    Mutex_lock(&M1);
    CriticalSection();
    Mutex_unlock(&M1);
    Mutex_unlock(&M2);
}

Suppose thread 1 is running and locks M1, but before it can lock M2, it is interrupted. Thread 2 starts running; it locks M2, when it tries to obtain and lock M1, it is blocked because M1 is already locked (by thread 1). Eventually, thread 1 starts running again, and it tries to obtain and lock M2, but it is blocked because M2 is already locked by thread 2. Both threads are blocked; each is waiting for an event that will never occur.

The deadlock situation in the above code can be modelled like this.

![Diagram showing deadlock between two threads and two mutexes M1 and M2.]

Thread 1

M1

Thread 2

M2
1. Is it possible to have a deadlock involving only one single process? Justify your answer.

No. This follows directly from the hold-and-wait condition, that is, with a single process, it is impossible that the hold and wait condition exists.

2. Recall the following example in class:
   - 5 processes are running in the system; $P_0$ through $P_4$.
   - They are using 3 resource types $A$ (with 10 instances), $B$ (with 5 instances), and $C$ (with 7 instances).
   - At time $T_0$, the snapshot of data structures maintained by the OS are as follows:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation Matrix</th>
<th>Max Matrix</th>
<th>Need Matrix</th>
<th>Available Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

(a) Can request vector for (3,3,0) by $P_4$ be granted?

(b) Can request vector for (0,2,0) by $P_0$ be granted?
3. Given the following system state that defines how 4 types of resources are allocated to 5 running processes.

\[
\text{Available} = \{2, 1, 0, 0 \} \\
\]

<table>
<thead>
<tr>
<th></th>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\text{Allocation} = \begin{pmatrix}
P0 & P1 & P2 & P3 & P0 \\
0 & 3 & 3 & 2 & 0 \\
\end{pmatrix} \quad \text{Max} = \begin{pmatrix}
0 & 0 & 1 & 2 & 0 \\
2 & 7 & 5 & 0 & 6 \\
6 & 6 & 5 & 6 & 4 \\
4 & 3 & 5 & 6 & 0 \\
6 & 5 & 2 & 0 & 6 \\
\end{pmatrix}
\]

a) Complete the Need matrix:

\[
\text{Need} = \begin{pmatrix}
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\end{pmatrix}
\]

b) Is the system in a safe state? Why or why not? If in a safe state, give a safe sequence.

c) For each of the following requests:

P0: Request = \{0, 1, 0, 0\}
P1: Request = \{0, 1, 0, 0\}
P2: Request = \{0, 1, 0, 0\}
P3: Request = \{0, 0, 0, 1\}

determine if the request should be granted. If it can be granted, show that the system is in a safe and give a safe sequence.
4. Deadlock Detection: Given the following system resource allocation state:

\[
\text{Available} = \{2, 1, 0, 0\} \\
\text{Allocation} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \\
\text{Request} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \\
\]

a) Determine if a deadlock exists.

b) Illustrate the system with a resource allocation graph.
Deadlock detection

If there is only one instance of each resource, it is possible to detect deadlock by constructing a resource allocation/request graph and checking for cycles. Graph theorists have developed a number of algorithms to detect cycles in a graph.

A cycle detection algorithm

For each node N in the graph
1. Initialize L to the empty list and designate all edges as unmarked
2. Add the current node to L and check to see if it appears twice. If it does, there is a cycle in the graph.
3. From the given node, check to see if there are any unmarked outgoing edges. If yes, go to the next step, if no, skip the next step
4. Pick an unmarked edge, mark it, then follow it to the new current node and go to step 3.
5. We have reached a dead end. Go back to the previous node and make that the current node. If the current node is the starting Node and there are no unmarked edges, there are no cycles in the graph. Otherwise, go to step 3.

Let's work through an example with five processes and five resources. Here is the resource request/allocation graph.

The algorithm needs to search each node; let's start at node P1. We add P1 to L and follow the only edge to R1, marking that edge. R1 is now the current node so we add that to L, checking to confirm that it is not already in L. We then follow the unmarked edge to P2, marking the edge, and making P2 the current node. We add P2 to L, checking to make sure that it is not already in L, and follow the edge to R2. This makes R2 the current node, so we add it to L, checking to make sure that it is not already there. We are now at a dead end so we back up, making P2 the current node again. There are no more
unmarked edges from P2 so we back up yet again, making R1 the current node. There are no more unmarked edges from R1 so we back up yet again, making P1 the current node. Since there are no more unmarked edges from P1 and since this was our starting point, we are through with this node (and all of the nodes visited so far).

We move to the next unvisited node P3, and initialize L to empty. We first follow the unmarked edge to R1, putting R1 on L. Continuing, we make P2 the current node and then R2. Since we are at a dead end, we repeatedly back up until P3 becomes the current node again.

L now contains P3, R1, P2, and R2. P3 is the current node, and it has another unmarked edge to R3. We make R3 the current node, add it to L, follow its edge to P4. We repeat this process, visiting R4, then P5, then R5, then P3. When we visit P3 again we note that it is already on L, so we have detected a cycle, meaning that there is a deadlock situation.

Sources:
http://www.cs.rpi.edu/academics/courses/fall04/os/e10/
http://www.site.uottawa.ca/~nelkadri/CSI3131/Tutorials/Tutorial%206/