- Deadlocks in a network
Consider a system of $n$ processes and $m$ different types of resources. Let us define the following vectors and matrices:

\[
\text{Resource} = (R_1, R_2, \ldots, R_m)
\]

total amount of each resource in the system

\[
\text{Available} = (V_1, V_2, \ldots, V_m)
\]

total amount of each resource not allocated to a process

\[
\text{Claim} = \begin{pmatrix}
C_{11} & C_{12} & \cdots & C_{1m} \\
C_{21} & C_{22} & \cdots & C_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & \cdots & C_{nm}
\end{pmatrix}
\]

requirement of each process for each resource

\[
\text{Allocation} = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1m} \\
A_{21} & A_{22} & \cdots & A_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
A_{n1} & A_{n2} & \cdots & A_{nm}
\end{pmatrix}
\]

current allocation
Figure 6.8  Determination of an Unsafe State
Figure 6.7. Determination of a Safe State
Figure 6.9  Example for Deadlock Detection

We can use Figure 6.9 to illustrate the deadlock detection algorithm. The algorithm proceeds as follows:

1. Mark P4, because P4 has no allocated resources.
2. Set $W = (0 0 0 0 1)$.
3. The request of process P3 is less than or equal to $W$, so mark P3 and set $W = W + (0 0 0 1 0) = (0 0 0 1 1)$.
4. No other unmarked process has a row in $Q$ that is less than or equal to $W$. Therefore, terminate the algorithm.

The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.
Figure 6.11  Dining Arrangement for Philosophers
/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}

Figure 6.12  A First Solution to the Dining Philosophers Problem

/* program diningphilosophers */
semaphore fork [5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}

Figure 6.13  A Second Solution to the Dining Philosophers Problem