- Messages received by C1 from C6, C7 and destined for C2 are buffered in a queue Q1.
- Messages received by C2 from C3, C4 and destined for C1 are buffered in a queue Q2.
- As traffic increases, Q1, Q2 will be full.
- Consequently,
  C1, C2 can't accept more messages.
  The communication path between C1, C2 becomes deadlocked.
6. Starvation

- The dining philosophers problem (Dijkstra, 1968)

Figure 6.10  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
7. Process Synchronization

- OS must make a resource unavailable to other processes while it is being used by one of them. Only when the resource is released is a waiting process allowed to use the resource. Process synchronization is critical here.

- The common element in all synchronization schemes is to allow a process to finish work on a critical region of the program before other processes have access to it.

- Synchronization is usually implemented as a lock-and-key arrangement: (1) the process must first see if the key is available and (2) if it is available, it must pick it up and put it in the lock to make it unavailable to other processes.

  - TEST-AND-SET (IBM 360/370): in a single CPU cycle it tests to see if the key is available and if it is, sets it to "unavailable".

  - WAIT-AND-SIGNAL: based on TEST-AND-SET, designed to remove busy waiting.

  - Semaphore: a nonnegative integer variable that's used as a flag.
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 waiting

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.

\[ V(s) : s \leftarrow s + 1 \]

\[ P(s) : \begin{array}{l}
\text{If } s > 0 \text{ then } s \leftarrow s - 1 \\
\text{If } s = 0 \text{ then Wait}
\end{array} \]

Traditionally, P, V operations are used to enforce Mutual Exclusion. So s is usually called a **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></td>
<td></td>
<td>P1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P1</td>
<td>P (s)</td>
<td>P1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P1</td>
<td>V (s)</td>
<td>P2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P2</td>
<td>P (s)</td>
<td>P2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P3</td>
<td>P (s)</td>
<td>P2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P4</td>
<td>P (s)</td>
<td>P3, P4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P2</td>
<td>V (s)</td>
<td>P3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>P3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>P3</td>
<td>V (s)</td>
<td>P4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P4</td>
<td>V (s)</td>
<td>P4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
8. Process Cooperation

- In real life, we have occasions when several processes work directly together to complete a common task. This is still the research topic of people in distributed computing.
  **Example:** *Several people try to edit a file over the Internet.*

- Producers and Consumers

- Readers and Writers
Producers and Consumers

![Diagram showing buffer states: (a) full, (b) half-full, (c) empty.]

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
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)

\( P (x): \; \text{if } x > 0 \text{ then } x = x - 1 \)

**CODEBEGIN** and **CODEEND** 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

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

```
\{
    PRODUCER: V (full) : full := 1
    // produce data
\}

\{
    Consumer: P (full) : full := 0
    // consume data
\}

Consumer: P (full) : Wait
// consumer wants to consume data, but has to wait
// as there is nothing available
```
 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).
4. Threads

- Multithreading — some modern operating systems support multiple threads of execution within a single process.

- In a multithread environment, a process is defined as the unit of protection and the unit of resource allocation.
  - Each process contains a virtual address space that holds the information related to it.
  - Each process contains protected access to processors, other processes, files and I/O devices.

- Benefits of using threads — *Performance*: it takes less time to create a new thread than to create a new process.
Figure 4.1
Threads and Processes [ANDE97]
Figure 4.2  Single Threaded and Multithreaded Process Models
• Given a process, there might be several threads, each with the following:
  
  – A thread execution state
  – A saved thread context when not running
  – An execution stack
  – Some pre-thread static storage for local variables
  – Access to the memory and resources of its process (threads in one process all have the same access)

• Thread states:
  
  – Spawn
  – Block (wait)
  – Unblock (ready)
  – Finish
Figure 4.3 Remote Procedure Call Using Threads
Figure 4.4 Multithreading Example on a Uniprocessor
5. Symmetric Multiprocessing

- Flynn proposed the following categories of computer systems in 1972.
  - Single Instruction Single Data (SISD) stream. Example, a PC runs MS-DOS.
  - Single Instruction Multiple Data (SIMD). Example, an array of processors doing numeric computations.
  - Multiple Instruction Single Data (MISD) stream. *Never been implemented.*
  - Multiple Instruction Multiple Data (MIMD) stream. *Most commonly used in practice.*

- In a symmetric multiprocessor the system kernel can execute on any processor.

- Clusters (Coarse Grained Multicomputers, Coarse Grained Multiprocessors) received a lot of attention since 1990s. Examples: PVM (Parallel Virtual Machine) and MPI (Massive Parallel Interface).
Figure 4.7 Parallel Processor Architectures
Figure 4.8  Symmetric Multiprocessor Organization