The semaphore which takes on any integer value is called a *counting semaphore*.
The semaphore which takes on any integer value is called a *counting semaphore*. An easier to implement version is a *binary semaphore* which ranges over only 0 and 1.
The semaphore which takes on any integer value is called a *counting semaphore*.

An easier to implement version is a *binary semaphore* which ranges over only 0 and 1.

The counting semaphore can be simulated using a combination of three binary semaphores $S_1, S_2, S_3$ and an integer variable $C$. 
Initialise \( S1 = S3 = 1 \) and \( S2 = 0 \).

\[
\text{wait}(S3);
\]

\[
\text{wait}(S1);
\]

\[
C \leftarrow C - 1;
\]

\[
\text{if } C < 0
\]

\[
\text{then begin}
\]

\[
\text{signal}(S1);
\]

\[
\text{wait}(S2);
\]

\[
\text{end}
\]

\[
\text{else signal}(S1);
\]

\[
\text{signal}(S3);
\]
signal operation using binary semaphores

\begin{verbatim}
wait(S1);
C ← C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
\end{verbatim}
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
- There is a common datastructure which is either modified or accessed by various users/processes.
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
- There is a common data structure which is either modified or accessed by various users/processes.
- Those who access to read are called *readers* and those who modify are called *writers*. 
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
- There is a common datastructure which is either modified or accessed by various users/processes.
- Those who access to read are called *readers* and those who modify are called *writers*.
- There is no problem with multiple readers accessing the file.
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
- There is a common data structure which is either modified or accessed by various users/processes.
- Those who access to read are called *readers* and those who modify are called *writers*.
- There is no problem with multiple readers accessing the file.
- There is however a problem if multiple writers access the file simultaneously, and also if a reader accesses the file at the same time as a writer.
The Readers and Writers problem

- This is a basic way to model the data inconsistency problem.
- There is a common data structure which is either modified or accessed by various users/processes.
- Those who access to read are called *readers* and those who modify are called *writers*.
- There is no problem with multiple readers accessing the file.
- There is however a problem if multiple writers access the file simultaneously, and also if a reader accesses the file at the same time as a writer.
- This is a standard critical section problem, with multiple variants based on fairness criteria.
Two variants

- **writer-friendly variant:** If a writer requests access, then no more new readers can access the file until the writer has been granted access and released the file.
Two variants

- **writer-friendly variant**: If a writer requests access, then no more new readers can access the file until the writer has been granted access and released the file.
- **reader-friendly variant**: Unless a writer has already got access to the file, no reader needs to wait for other readers to finish reading.
Two variants

- **writer-friendly variant:** If a writer requests access, then no more new readers can access the file until the writer has been granted access and released the file.

- **reader-friendly variant:** Unless a writer has already got access to the file, no reader needs to wait for other readers to finish reading.

  The first leads to reader starvation and the second to writer starvation.
There are semaphores \textit{mutex}, \textit{wrt} and an integer variable \textit{readcount} which maintains the current number of readers.

\textbf{Structure of a writer process:}

\texttt{wait(wrt);}  \\
\hspace{1cm}\text{WRITING SECTION}  \\
\texttt{signal(wrt);} \\

\textbf{Structure of a reader process:}

\texttt{wait(mutex);}  \\
\hspace{1cm} \texttt{readcount \leftarrow readcount + 1;}  \\
\hspace{1cm} \texttt{if readcount = 1 then wait(wrt);}  \\
\texttt{signal(mutex);}  \\
\hspace{1cm}\text{READING SECTION}  \\
\texttt{wait(mutex);}  \\
\hspace{1cm} \texttt{readcount \leftarrow readcount - 1;}  \\
\hspace{1cm} \texttt{if readcount = 0 then signal(wrt);}  \\
\texttt{signal(mutex);}
Programming errors when using semaphores

- If the *signal* or *wait* operations are interchanged, or one of them is missing or is duplicated, it can lead to violations of one or more requirements of the critical section protocol.
If the *signal* or *wait* operations are interchanged, or one of them is missing or is duplicated, it can lead to violations of one or more requirements of the critical section protocol.

This kind of a mistake is easy to make when coding a lot of processes.
Programming errors when using semaphores

- If the *signal* or *wait* operations are interchanged, or one of them is missing or is duplicated, it can lead to violations of one or more requirements of the critical section protocol.
- This kind of a mistake is easy to make when coding a lot of processes.
- Such errors are not always detectable by testing, because the runs are not identical, so an error which does not occur in one trial may occur later.
If the *signal* or *wait* operations are interchanged, or one of them is missing or is duplicated, it can lead to violations of one or more requirements of the critical section protocol.

This kind of a mistake is easy to make when coding a lot of processes.

Such errors are not always detectable by testing, because the runs are not identical, so an error which does not occur in one trial may occur later.

One possible solution to reduce the chances of this type of mistake is the *critical regions* construct.
Critical Regions construct

- This is based on using *boolean* or *logical* statements to control access to critical sections. These are often clearer and easier for a programmer to understand and use than a semaphore.
This is based on using *boolean* or *logical* statements to control access to critical sections. These are often clearer and easier for a programmer to understand and use than a semaphore.

The general statement is: `region ν when B do S;`
This is based on using boolean or logical statements to control access to critical sections. These are often clearer and easier for a programmer to understand and use than a semaphore.

The general statement is: \texttt{region \textit{v} when \textit{B} do \textit{S};}

Here, \textit{B} is a boolean condition which controls access to the critical region. It is specific to the variable \textit{v}, and ensures that no other process associated with \textit{v} can execute, when it is in this region (executing statement \textit{S}).
wait(mutex);
while not B
do begin
  first-count ← first-count + 1;
  if second-count > 0
    then signal(second-delay)
    else signal(mutex);
  wait(first-delay); first-count ← first-count − 1;
  second-count ← second-count + 1;
  if first-count > 0
    then signal(first-delay)
    else signal(second-delay);
  wait(second-delay);
  second-count ← second-count − 1;
end
$S$

if first-count > 0
    then signal(first-delay);
else if second-count > 0
    then signal(second-delay);
else signal(mutex);
type monitor-name = monitor
  
  variable declarations

procedure entry P1(...);
  begin...end;

begin
  initialisation code
end;