```
monitor dining_controller;
cond ForkReady[5];                /* condition variable for synchronization */
boolean fork[5] = {true};                /* availability status of each fork */
void get_forks(int pid)                  /* pid is the philosopher id number */
{
   int left = pid;
   int right = (pid++) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);                /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady[right]);               /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (pid++) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])                /*no one is waiting for this fork */
      fork(left) = true;
   else   /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])               /*no one is waiting for this fork */
      fork(right) = true;
   else                              /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]                   /* the five philosopher clients */
{
   while (true)
   {
      <think>;
      get_forks(k);                    /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);                /* client releases forks via the monitor */
   }
}
```

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 Waiti

WAIT-and-SIGNAL

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

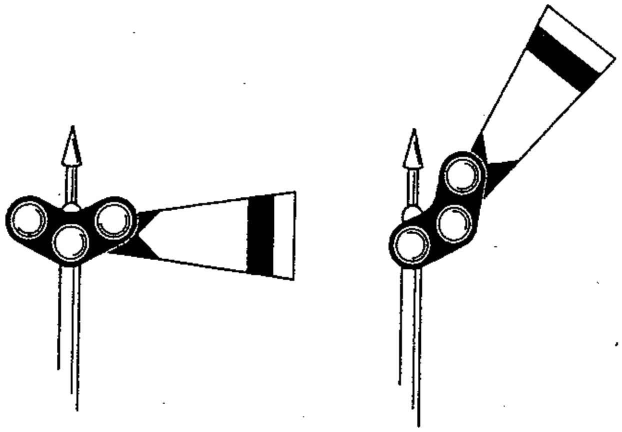
WAIT is activated when the process encounters a busy condition code.

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): $s \leftarrow s+1$

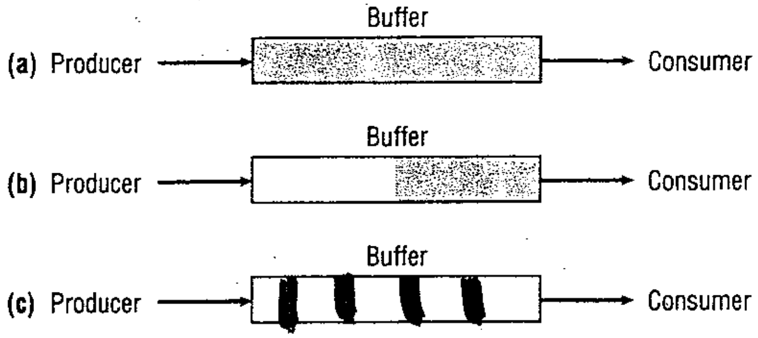
P(s): If $s>0$ then $s \leftarrow s-1$
If $s=0$ then wait

Traditionally, P.V operations are used to enforce Mutual Exclusion. So s is usually called **mutex.**

The sequence of states for four processes calling P and V operations on the binary semaphore s. (Note: the value of the semaphore before the operation is on the line preceding the operation. The current value is on the same line.)

Actions			Results		
State number	Calling process	Operation	Running in critical region	Blocked on *s*	Value of *s*
0					1
1	P1	P (*s*)	P1		0
2	P1	V (*s*)			1
3	P2	P (*s*)	P2		0
4	P3	P (*s*)	P2	P3	0
5	P4	P (*s*)	P2	P3, P4	0
6	P2	V (*s*)	P3	P4	0
7			P3	P4	0
8	P3	V (*s*)	P4		0
9	P4	V (*s*)			1

Producers and Consumers



buffer full

buffer half full

buffer 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

The 3rd semaphore will ensure mutual exclusion.

3. Mutex

Here are the definitions of the producer and consumer processes:

PRODUCER	CONSUMER
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	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:

```
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

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.

<u>Solution 1</u>: Readers are kept waiting only if a writer is modifying the data.

problem?

Writer starvation

<u>Solution 2</u>: Once a writer arrives, readers that are active are allowed to finish processing, but all additional readers are put on hold.

Problem?

Reader Starvation

<u>Solution 3</u>:

- 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).