

Semaphores

Dekker's algorithm solves the mutual exclusion problem on a shared memory machine with no support from the hardware or software. *Semaphores* are a higher level concept than atomic instructions. They are atomic actions¹ and usually implemented within the operating system.

A semaphore S is a non-negative integer variable that has exactly two operations defined for it.

P(S) If $S > 0$ then $S = S - 1$, otherwise suspend the process.

V(S) If there are processes suspended on this semaphore wake one of them, else $S = S + 1$.

An important point is that V(S), as it is currently defined, does not specify which of the suspended processes to wake.

Semaphore Invariants

The following invariants are true for semaphores.

$$S \geq 0$$

$$S = S_0 + \#V - \#P$$

where S_0 is the initial value of semaphore S .

¹ Aka *coarse-grained* atomic action made up of a series of transactions executed without interruption.

Mutual Exclusion

```
sem mutex := 1
const N := 20

process p(i := 1 to N)
  do true ->
    Non_critical_Section
    P(mutex) # grab mutex semaphore
    Critical_Section
    V(mutex) # release mutex semaphore
  od
end
```

Theorem Mutual exclusion is satisfied.

Proof: Let #CS be the number of processes in their critical sections. We need to prove that the following is an invariant.

$$\#CS + \text{mutex} = 1$$

- i) $\#CS = \#P - \#V$;from the program structure
- ii) $\text{mutex} = 1 + \#V - \#P$;semaphore invariant
- iii) $\text{mutex} = 1 - \#CS$;from i) and ii)
- iv) $\#CS + \text{mutex} = 1$; from iii)

QED

Theorem The program cannot deadlock.

Proof: This would require all processes to be suspended in their P(mutex) operations. Then mutex = 0 and #CS = 0 since no process is in its critical section. The critical section invariant just proven is :

$$\#CS + \text{mutex} = 1 \Rightarrow 0 + 0 = 1$$

which is impossible.

Types of Semaphores

What we defined earlier is a general semaphore. A *binary semaphore* is a semaphore that can only take the values 0 and 1.

The choice of suspended process to wake gives the following definition.

Blocked-set semaphore Awakens any one of the suspended processes.

Blocked-queue semaphore The suspended processes are kept in FIFO and are awakened in the same order they were suspended. This is the type of semaphore implemented in the SR language.

Busy-wait semaphore The value of the semaphore is tested in a busy wait loop, with the test being atomic. There may be interleavings between cycles of the loop.

Theorem With busy-wait semaphores, starvation is possible.

Proof: Consider the following execution sequence for 2 processes.

- i) P1 executes P(mutex) and enters its critical section.
- ii) P2 executes P(mutex), finds mutex = 0 and loops.
- iii) P1 finishes critical section, executes V(mutex) loops back and execute P(mutex) and enters its critical section.
- iv) P2 test mutex, finds mutex = 0, and loops.

Theorem With blocked-queue semaphores, starvation is impossible.

Proof: If P1 is blocked on mutex there will be at most N-1 processes ahead of P1 in the queue. Therefore after N-1 V(mutex) P1 will enter its critical section.

Theorem With blocked-set semaphores, starvation is possible for $N \geq 3$.

Proof: If there are 3 processes it is possible to construct an execution sequence such that there are always 2 processes blocked on a semaphore. V(mutex) is required to only wake one of them, so it could always ignore one and leave that process starved.

The Producer-Consumer Problem

This type of problem has two types of processes:

Producers processes that, due to some internal activity, produce data to be sent to consumers.

Consumers processes that on receipt of a data element consume the data in some internal computation.

We could connect these processes in a synchronous manner, such that data is only transmitted when a producer is ready to send it and a consumer is ready to receive it. A more flexible method is to connect the producers and the consumers by a buffer which is a *queue*. `Produce()`, `Consume()` operations are assumed.

If this was an infinite buffer then the following invariants should hold true for the buffer.

$$\#elements \geq 0$$

$$\#elements = 0 + in_pointer - out_pointer$$

These invariants are exactly the same as the semaphore invariants with a semaphore called *elements* and an initial value 0.

```

var buffer [?]:int
var in_pointer:int := 0,
out_pointer:int := 0
sem elements := 0

process producer
  do true ->
    buffer [in_pointer] := produce ()
    in_pointer := in_pointer + 1
    V(elements)
  od
end

process consumer
var i:int
  do true ->
    P (elements)
    i := buffer [out_pointer]
    out_pointer := out_pointer + 1
    consume (i)
  od
end

```

This can be modified for real bounded circular buffers by using another semaphore to count the empty places in the buffer.

As an exercise prove the following:

(i) the program cannot deadlock, (ii) neither process is starved and (iii) the program never removes data from an empty buffer or appends data to a full buffer.

```

const N := 100
var buffer [N]:int
var in_pointer:int := 0,
out_pointer:int := 0
sem elements := 0
sem spaces := N

process producer
var i:int
  do true ->
    i := produce ()
    P (spaces)
    buffer [in_pointer] := i
    in_pointer := (in_pointer+1) mod N
    V (elements)
  od
end

process consumer
var i:int
  do true ->
    P (elements)
    i := buffer [out_pointer]
    out_pointer := (out_pointer+1) mod N
    V (spaces)
    consume (i)
  od
end

```

Circular Buffer Proofs from above

Semaphore Invariants:

#E=elements (No. of elements in Buffer=Value of elements Semaphore)

#E = N - Spaces

1. No Deadlock

- For deadlock, both processes are suspended on semaphores so spaces= elements= 0. From the invariants above, this is impossible.

2. No Starvation

- Assume producer suspended on P(spaces) indefinitely while consumer executes an infinite number of cycles of its loop.
- But even a single cycle of consumer loop causes a V(spaces), releasing the producer.

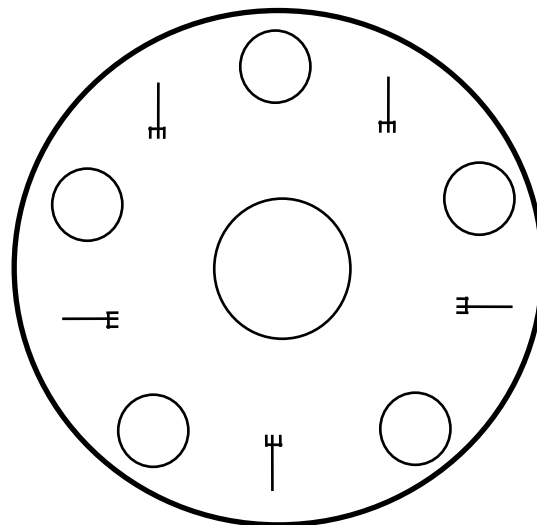
3. Program never removes an element from an empty buffer

- Taking from empty buffer is equiv to executing cycle of consumer loop while #E=0
- If #E=0 consumer will be suspended on P(elements) and cannot execute this.

The Dining Philosophers Problem

The Problem

An institution hires five philosophers to solve a very difficult problem. Each philosopher only engages in two activities - *thinking* and *eating*. Meals are taken in the diningroom which has a table set with five plates and five forks. In the centre of the table is a bowl of spaghetti that is endlessly replenished. The philosophers, not being the most dextrous of individuals, requires two forks to eat; and may only pick up the forks immediately to his left and right.



For this system to operate correctly it is required that:

- a) A philosopher eats only if he has two forks.
- b) No two philosophers can hold the same fork simultaneously.
- c) There can be no deadlock.

- d) There can be no individual starvation.
- e) There must be efficient behaviour under the absence of contention.

This problem is a generalisation of multiple processes accessing a set of shared resources; e.g. a network of computers accessing a bank of printers.

First Attempted Solution

Model each fork as a semaphore. Then each philosopher must wait (execute a P operation) on both the left and right forks before eating.

```
sem fork [5] := ([5] 1)
# fork is array of semaphores
# all initialised to have value 1
process philosopher (i := 0 to 4)
do true ->
  Think ( )
  P(fork [i]) #grab fork[i]
  P(fork [(i+1) mod 5] #grab rh fork
  Eat ( )
  V(fork [i]) #release fork[i]
  V(fork [(i+1) mod 5] #and rh fork
od
end
```

This is called a *symmetric solution* since each task is identical. Symmetric solutions have many advantages, particularly when it comes to load-balancing.

We can prove that no fork is ever held by two philosophers since `Eat ()` is the critical section of each fork. If $\#P_i$ is the number of philosophers holding fork i then we have:

$$Fork(i) + \#P_i = 1$$

(ie either philosopher is holding the fork or sem is 1)

Since a semaphore is non-negative then $\#P_i \leq 1$.

However, this system can deadlock (i.e none can eat) under an interleaving in which all five philosophers pick up their left forks together; i.e. all processes execute `P(fork [i])` before `P(fork [(i+1) mod 5])`. There are two solutions. One is to make one of the philosophers pick up the right fork before the left fork (asymmetric solution); the other is to only allow four philosophers into the room at any one time.

A Symmetric Solution

```
sem Room := 4
sem fork [5] := ([5] 1)
Process philosopher (i := 0 to 4)
  do true ->
    Think ( ) # thinking not a CS!
    P (Room)
    P(fork [i])
    P(fork [(i+1) mod 5])
    Eat ( ) # eating is the CS
    V(fork [i])
    V(fork [(i+1) mod 5])
    V (Room)
  od
end
```

This solves the deadlock problem.

Theorem Individual starvation cannot occur.

Proof For a process to starve it must be forever blocked on one of the three semaphores, Room, fork [i] or fork [(i+1) mod 5].

Room semaphore

If the semaphore is a blocked-queue semaphore then process i is blocked only if Room is 0 indefinitely. This would require the other four philosophers to be blocked on their left forks, since if one of them can get two forks he will finish, put down the forks and signal Room (by $V(\text{Room})$). So this case will follow from the fork [i] case.

fork [i]

If philosopher i is blocked on his left fork, then philosopher $i-1$ must be holding his right fork. Therefore he is either eating or signalling he is finished with his left fork, and will eventually put down his right fork which is philosopher i 's left fork.

fork [(i+1) mod 5]

If philosopher i is blocked on his right fork, this means that philosopher $(i+1)$ has taken his left fork and never released it. Since eating and signalling cannot block, philosopher $(i+1)$ must be waiting for his right fork, and must all the other philosophers by induction: $i+j$, $0 \leq j \leq 4$. However the Room semaphore invariant only 4 philosophers can be in the room, so philosopher i cannot be blocked on his right fork.

Readers-Writers Problem

The Problem

Two kinds of processes, readers and writers, share a database. Readers execute transactions that examine the database, writers execute transactions that examine and update the database. Given that the database is initially consistent, then to ensure that the database remains consistent, a writer process must have exclusive access

to the database. Any number of readers may concurrently examine the database.

Obviously, as far as a writer process is concerned, updating the database is a critical section that cannot be interleaved with any other process.

```
const M:int := 20, N:int := 5
var nr:int := 0
sem mutexR := 1
sem rw := 1

process reader (i:= 1 to M)
do true ->
  P (mutexR)
  nr := nr + 1
  if nr = 1 -> P (rw) fi
  V (mutexR)
  Read_Database ( )
  P (mutexR)
  nr := nr - 1
  if nr = 0 -> V (rw) fi
  V (mutexR)
od
end

process writer(i:=1 to N)
do true ->
  P (rw)
  Update_Database ( )
  V (rw)
od
end
```

This is called the *readers' preference* solution since if some reader process is accessing the database and a reader process and a writer process arrive at their entry

protocols then the reader process will always have preference over the writer process.

This is not a fair solution since this solution always gives readers precedence over writers, a continual stream of readers will block any writer process from updating the database.

To make the solution fair we need to use a *split binary semaphore*, that is several semaphores who together have the property that their sum is 0 or 1. We also need to count the number of suspended reader processes and suspended writer processes.

This technique is called *passing the baton*.

Readers-Writers: Passing the Baton

```

const M:int := 20, N:int := 5
var nr:int :=0, nw:int := 0
var sr:int := 0, sw:int := 0
  # count of suspended
  # readers and writers
sem e := 1, r := 0, w := 0
  #  $0 \leq (e+r+w) \leq 1$ 

```

```

process reader (i:= 1 to M)
do true ->
  P (e)
  {
  if nw > 0 ->
    sr:= sr + 1; V(e); P(r)
  fi
  nr := nr + 1
  {
  if sr > 0 ->
    sr := sr - 1; V (r)
  [] sr = 0 -> V(e)
  fi
  Read_Database ( )

  P (e)
  nr := nr - 1
  {
  if nr = 0 and sw > 0 ->
    sw := sw - 1; V (w)
  [] nr >0 or sw = 0 -> V(e)
  fi
  od
end

```

```

process writer (i:= 1 to N)
do true ->
  P (e)
  {
  if nr > 0 or nw > 0 ->
    sw:= sw + 1; V(e); P(w)
  fi
  nw := nw + 1
  V (e)

  Update_Database ( )

  P (e)
  nw := nw - 1
  {
  if sr >0 -> sr:= sr-1;V(r)
  [] sw >0 -> sw:= sw-1;V(w)
  [] sr =0 and sw =0 -> V(e)
  fi
  od
end

```

Passing the Baton:

Called 'Passing the Baton' because of way signalling takes place (when a process is executing within a CS, it holds the 'baton'). When that process gets to an exit point from that CS, it 'passes the baton' to some other process. If ($>$) one process is waiting for a condition that is now true, 'baton is passed' to one such process, randomly. If none is waiting, baton is passed to next one trying to enter the CS for the first time, ie trying $P(e)$.

A Number of Possible Scenarios:

1. Suppose a writer is in first....
 - Any readers executing $P(e)$ will be suspended in a FIFO queue ($sr := sr + 1$)
 - The writer will finish, execute $P(e)$, decrement n_w and eventually signal a suspended (or maybe a new) reader who can then increment n_r , awake the sr .
 - Note that the if in SR is non-deterministic (any of the $[]$ arms which apply can be executed non-deterministically)
2. Suppose a reader is first to grab the entry semaphore....
 - More readers can possibly be let in since there are no sr 's ($[] sr=0 \rightarrow V(e)$)
 - A writer can come in but is immediately suspended pending the signal from the last reader to exit after reading the database
 - Note that (as above) the if at the end of process reader is non-deterministic

Ballhausen's solution to Readers-Writers Problem

The idea behind this solution is one of efficiency: a reader takes up the same space as all readers reading together. A semaphore access is used for readers gaining entry to the database, with a value initially equalling the total number of readers. Every time a reader accesses the database, the value of this semaphore is decremented and when one leaves, it is incremented. When a writer wants to enter the database it will occupy all space step by step by waiting for all old readers to leave and blocking entry to new ones. The writer uses a semaphore mutex to prevent deadlock between two writers trying to occupy half of the available space each.

```
sem mutex = 1
sem access = m

process reader (i = 1 to m)
  do true ->
    P(access)
    # ... reading ...
    V(access)
    # other operations
  od
end

process writer (j = 1 to n)
  do true ->
    P(mutex)
    fa k = 1 to m ->
      P(access) af
      # ... writing ...
    fa k = 1 to m ->
      V(access) af
      # other operations
    od
  end
end
```

Monitors

The main disadvantage with semaphores is that they are a low level programming construct. In a large project, with many programmers, if one programmer forgets to do V() operation on a semaphore after a critical section, then the whole system can deadlock.

What is required is a higher level construct that groups the responsibility for correctness into a few modules.

Monitors are such a construct. These are an extension of the monolithic monitor found in operating systems for allocating memory etc. They *encapsulate* a set of procedures, and the data they operate on, into single modules, called monitors, and guarantee that only one process can execute a procedure in the monitor at any given time (mutual exclusion). Of course different processes can execute procedures from different monitors at the same time.

Condition Variables

Synchronisation is achieved by using *condition variables*. These are data structures that have 3 operations defined for them.

- wait (C)** The process that called the monitor containing this operation is suspended in a FIFO queue associated with C. Mutual exclusion on the monitor is released.
- signal (C)** If the queue associated with C is non-empty, then wake the process at the head of the queue.
- non-empty (C)** Returns true if the queue associated with C is non-empty.

Note the difference between the P in semaphores and wait(C) in monitors: latter always delays until signal(C) is called, former only if the semaphore variable is zero.

If a monitor guarantees mutual exclusion, and a process uses the *signal* operation and awakens another process suspended in the monitor, is there not two processes in the same monitor at the same time? Yes.

To solve this problem, several signalling mechanisms can be implemented, the simplest of which is the *signal and continue mechanism*. Under these rules the procedure in the monitor that signals a condition variable is allowed to continue to completion, so the *signal* operation should be at the end of the procedure. The process which was suspended on the condition variable, but is now awoken, is scheduled for *immediate resumption* on the exiting of procedure which signalled the condition variable.

Readers/Writers Problem: Monitor Solution

```
_monitor (RW_control)
  op request_read ( )
  op release_read ( )
  op request_write ( )
  release_write ( )
_body (RW_control)
  var nr:int := 0, nw:int := 0
  _condvar (ok_to_read)
  _condvar (ok_to_write)

  _proc (request_read ( ))
    do nw > 0 ->
      _wait (ok_to_read) od
    nr := nr + 1
  _proc_end

  _proc (release_read ( ))
    nr := nr - 1
    if nr = 0 ->
      _signal(ok_to_write) fi
  _proc_end

  _proc (request_write ( ))
    do nr > 0 or nw > 0 ->
      _wait (ok_to_write)
    od
    nw := nw + 1
  _proc_end

  _proc (release_write ( ))
    nw := nw - 1
    _signal (ok_to_write)
    _signal_all (ok_to_read)
  _proc_end
_monitor_end

resource main ( )
  import RW_control

  process reader (i:= 1 to 20)
    RW_control.request_read( )
    Read_Database ( )
    RW_control.release_read( )
  end

  process writer (i := 1 to 5)
    RW_control.request_write()
    Update_Database ( )
    RW_control.release_write()
  end
end
```

File rw_control.m

Emulating Semaphores with Monitors

Both semaphores and monitors are concurrent programming primitives of equal power. Monitors are just a higher level construct.

```
_monitor semaphore
  op p ( ), v ( )
_body semaphore
  var s:int := 0
  _condvar (not_zero)
  _proc (p ( ))
    if s=0 ->
      _wait (not_zero) fi # only _wait if s=0
    s := s - 1
  _proc_end

  _proc (v ( ))
    if _not_empty (not_zero) = true ->
      _signal (not_zero) # only _signal if
                        # suspended processes
    [] else ->          # else increment s
      s := s + 1
    fi
  _proc_end
_monitor_end
```

Emulating Monitors by Semaphores

Firstly we need blocked-queue semaphores (SR is OK) and secondly we need to implement the *signal and continue* mechanism. We do this with a variable `c_count`, one semaphore, `s`, to ensure mutual

exclusion and another semaphore, `c_semaphore`, to act as the condition variable. Then we translate `_wait (s)` to:

```
c_count := c_count + 1
V (s)
P (c_semaphore) #_wait always suspends
c_count := c_count - 1
                # 1 less process in monitor
```

and `_signal (s)` to:

```
if c_count > 0 ->
    V (c_semaphore) # only _signal if
                   #waiting processes
[] else -> V (s)
fi
```

Monitor entry, exit implemented by `P (s)`, `V (s)` resp.

The Dining Philosophers Problem with Monitors

Using monitors yields a nice solution, since with semaphores you cannot test two semaphores simultaneously. The monitor solution maintains an array `fork` which counts the number of free forks available to each philosopher.

```

_monitor (fork_mon)
  op take_fork (i:int),
  release_fork (i:int)

_body (fork_mon)
  var fork [5]:int := ([5] 2)
  _condvar (ok2eat, 5)
# define an array of
# condition variables

_proc (take_fork (i))
  if fork [i] != 2 ->
    _wait (ok2eat[i]) fi
  fork [(i-1) mod 5] :=
    fork [(i-1) mod 5] - 1
  fork [(i+1) mod 5] :=
    fork [(i+1) mod 5] - 1
_proc_end

_proc (release_fork (i))
  fork [(i-1) mod 5] :=
    fork [(i-1) mod 5] + 1
  fork [(i+1) mod 5] :=
    fork [(i+1) mod 5] + 1

  if fork[(i+1) mod 5]= 2 ->
_signal(ok2eat[(i+1) mod 5])
  fi  #rh phil can eat

  if fork[(i-1) mod 5]= 2 ->
_signal(ok2eat[(i-1) mod 5])
  fi  #lh phil can eat

_proc_end

_monitor_end

```

```

resource main ( )
  import fork_mon

  process philosopher (i:= 1
to 5)
  do true ->
    Think ( )
    fork_mon.take_fork (i)
    Eat ( )
    fork_mon.release_fork(i)
  od
end
end

```

Theorem Solution doesn't deadlock.

Proof: Let #E be the number of philosophers who are eating, and have therefore taken both forks. Then the following invariants are true from the program.

$$\text{Non-empty}(\text{ok2eat}[i]) \Rightarrow \text{fork}[i] < 2$$

$$\sum_{i=1}^5 \text{fork}[i] = 10 - 2(\#E)$$

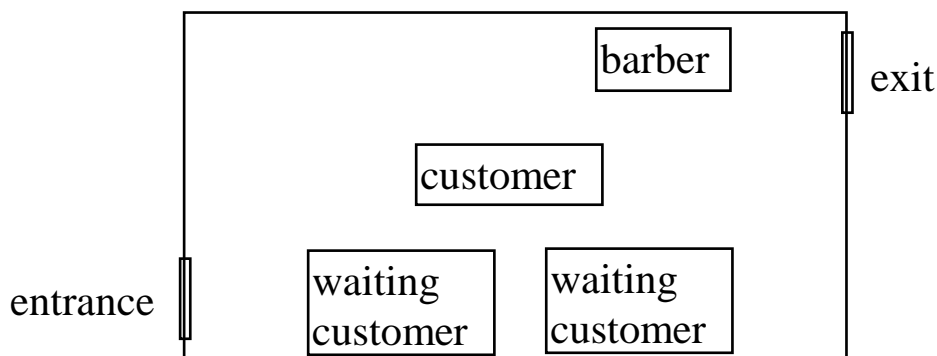
Deadlock implies #E=0 and all philosophers are enqueued on `ok2eat`. If they are all enqueued then the first equation implies $\sum \text{fork}[i] \leq 5$ and if no philosopher is eating, then the second equation implies $\sum \text{fork}[i] = 10$. The contradiction implies that the solution does not deadlock.

However, individual starvation can occur. How? What is the solution?

The Sleeping Barber Problem

This is a generalisation of *rendezvous* synchronisation in *client/server* architectures.

Problem A small barber shop has two doors, an entrance and an exit. Inside is a barber who spends all his life serving customers, one at a time. When there are none in the shop, he sleeps in his chair. If a customer arrives and finds the barber asleep he awakens the barber, sits in the customer's chair and sleeps while his hair is being cut. If a customer arrives and the barber is busy cutting hair, the customer goes asleep in one of the two waiting chairs. When the barber finishes cutting a customer's hair, he awakens the customer and holds the exit door open for him. If there are any waiting customers, he awakens one and waits for the customer to sit in the customer chair, otherwise he goes to sleep.



The barber and customers are interacting processes, and the barber shop is the monitor in which they react.

```

_monitor (barber_shop)
  op get_haircut ( ), get_next_customer ( )
  op finish_cut ( )

_body (barber_shop)
  var barber:int := 0
  var chair:int := 0, open:int := 0
  _condvar (barber_available) # when barber > 0
  _condvar (chair_occupied)   # when chair > 0
  _condvar (door_open)        # when open > 0
  _condvar (customer_left)    # when open = 0

# called by customer
_proc (get_haircut ( ))
  do barber = 0 -> _wait (barber_available) od
  barber := barber - 1
  chair := chair + 1
  _signal (chair_occupied)
  do open = 0 -> _wait (door_open) od
  open := open - 1
  _signal (customer_left)
_proc_end

# called by barber
_proc (get_next_customer ( ))
  barber := barber + 1
  _signal (barber_available)
  do chair = 0 -> _wait (chair_occupied) od
  chair := chair - 1
_proc_end

# called by barber
_proc (finished_cut ( ))
  open := open + 1
  _signal (door_open)
  do open > 0 -> _wait (customer_left) od
_proc_end
_monitor_end

```

```

resource main ( )
  import barber_shop

  process customer (i:= 1 to 5)
    do true ->
      barber_shop.get_haircut (i)
    od
  end
  process barber ( )
    do true ->
      barber_shop.get_nextcustomer ( )
      barber_shop.finished_cut ( )
    od
  end
end

```

In general, when many different processes have to *rendezvous*, the solution technique is to use a monitor to create an “environment” in which the processes can *rendezvous*.