```
type  semaphore =  record
                       count: integer;
                       queue: list of process
                   end;
var s: semaphore;

wait(s):
        s.count := s.count - 1;
        if s.count < 0
            then begin
                place this process in s.queue;
                block this process
            end;

signal(s):
        s.count := s.count + 1;
        if s.count ≤ 0
            then begin
                remove a process P from s.queue;
                place process P on ready list
            end;
```

Figure 5.8 A Definition of Semaphore Primitives

```
type  binary semaphore =     record
                               value: (0,1);
                               queue: list of process
                             end;
var s: binary semaphore;

waitB(s):
        if s.value =  1
           then
              s.value = 0
           else begin
              place this process in s.queue;
              block this process
           end;

signalB(s):
        if s.queue is empty
           then
              s.value := 1
           else begin
              remove a process P from s.queue;
              place process P on ready list
           end;
```

Figure 5.9 A Definition of Binary Semaphore Primitives

```
program      mutualexclusion;
const n = . . . ; (*number of processes*);
var      s: semaphore (:=1);
procedure P(i: integer);
begin
   repeat
   wait(s);
   < critical section >;
   signal(s);
   < remainder >
   forever
end;
begin (* main program *)
   parbegin
      P(1);
      P(2);
      . . .
      P(n)
   parend
end.
```

Figure 5.10 Mutual Exclusion Using Semaphores

```
program    producerconsumer;
var        n: semaphore (:= 0);
           s: semaphore (:= 1);
procedure producer;
begin
   repeat
      produce;
      wait(s);
      append;
      signal(s);
      signal(n)
   forever
end;
procedure consumer;
begin
   repeat
      wait(n);
      wait(s);
      take;
      signal(s);
      consume
   forever
end;
begin (*main program*)
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.15 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

```
program    producerconsumer;
var        n: integer;
           s: (*binary*) semaphore (:= 1);
           delay: (*binary*) semaphore (:= 0);
procedure producer;
begin
   repeat
      produce;
      waitB(s);
      append;
      n := n + 1;
      if n=1 then signalB(delay);
      signalB(s)
   forever
end;
procedure consumer;
begin
   waitB(delay);
   repeat
      waitB(s);
      take;
      n := n - 1;
      signalB(s);
      consume;
      if n=0 then waitB(delay)
   forever
end;
begin (*main program*)
   n := 0;
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.13 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
program     producerconsumer;
var         n: integer;
            s: (*binary*) semaphore (:= 1);
            delay: (*binary*) semaphore (:= 0);
procedure producer;
begin
    repeat
        produce;
        waitB(s);
        append;
        n := n + 1;
        if n=1 then signalB(delay);
        signalB(s)
    forever
end;
procedure consumer;
var     m: integer; (* a local variable *);
begin
    waitB(delay);
    repeat
        waitB(s);
        take;
        n := n - 1;
        m := n;
        signalB(s);
        consume;
        if m=0 then waitB(delay)
    forever
end;
begin (*main program*)
    n := 0;
    parbegin
        producer; consumer
    parend
end.
```

Figure 5.14 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
program    boundedbuffer;
const sizeofbuffer = . . .;
var          s:  semaphore (:= 1);
             n: semaphore (:= 0);
             e: semaphore (:= sizeofbuffer);
procedure producer;
begin
   repeat
      produce;
      wait(e);
      wait(s);
      append;
      signal(s);
      signal(n)
   forever
end;
procedure consumer;
begin
   repeat
      wait(n);
      wait(s);
      take;
      signal(s);
      signal(e);
      consume
   forever
end;
begin (*main program*)
   parbegin
      producer; consumer
   parend
end.
```

Figure 5.17 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

```
var     flag: array [0 .. 1] of boolean;
        turn: 0 .. 1;
procedure P0;
begin
    repeat
        flag [0] := true;
        while flag [1] do if turn = 1 then
                        begin
                            flag [0] := false;
                                while turn = 1 do { nothing };
                            flag [0] := true
                        end;
        < critical section >;
        turn := 1;
        flag [0] := false;
        < remainder >
    forever
end;
procedure P1;
begin
    repeat
        flag [1] := true;
        while flag [0] do if turn = 0 then
                        begin
                            flag [1] := false;
                                while turn = 0 do { nothing };
                            flag [1] := true
                        end;
        < critical section >;
        turn := 0;
        flag [1] := false;
        < remainder >
    forever
end;
begin
    flag [0] := false;
    flag [1] := false;
    turn := 1;
    parbegin
        P0; P1
    parend
end.
```

Figure 5.5 Dekker's Algorithm

```
var    flag: array [0 .. 1] of boolean;
       turn: 0 .. 1;
procedure P0;
begin
   repeat
      flag [0] := true;
      turn := 1;
      while flag [1] and turn = 1 do { nothing };
      < critical section >;
      flag [0] := false;
      < remainder >
   forever
end;
procedure P1;
begin
   repeat
      flag [1] := true;
      turn := 0;
      while flag [0] and turn = 0 do { nothing };
      < critical section >;
      flag [1] := false;
      < remainder >
   forever
end;
begin
   flag [0] := false;
   flag [1] := false;
   turn := 1;
   parbegin
      P0; P1
   parend
end.
```

Figure 5.6 Peterson's Algorithm for Two Processes

```
program      mutualexclusion;
const n = . . . ; (*number of processes*);
procedure P(i: integer);
begin
   repeat
      entercritical (R);
      < critical section >;
      exitcritical (R);
      < remainder >
   forever
end;
begin (* main program *)
   parbegin
      P(1);
      P(2);
      . . .
      P(n)
   parend
end.
```

Figure 5.1 Mutual Exclusion

```
const
    capacity = …;      {buffering capacity}
    null = …;          {empty message}
var    i: integer;
procedure producer;
    var pmsg: message;
    begin
        while true do
            begin
                receive (mayproduce, pmsg);
                pmsg := produce;
                send (mayconsume, pmsg)
            end
    end;
procedure consumer;
    var csmg: message;
    begin
        while true do
            begin
                receive (mayconsume, cmsg);
                consume (csmg);
                send (mayproduce, null)
            end
    end;

{parent process}
begin
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for i = 1 to capacity do send (mayproduce, null);
    parbegin
        producer;
        consumers
    parend
end
```

Figure 5.27 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

```
program readersandwriters;
var   readcount: integer;
      x, wsem: semaphore (:= 1);
procedure reader;
begin
   repeat
      wait (x);
         readcount := readcount + 1;
         if readcount = 1 then wait (wsem);
      signal (x);
      READUNIT;
      wait (x);
         readcount := readcount - 1;
         if readcount = 0 then signal (wsem);
      signal (x)
   forever
end;
procedure writer;
begin
   repeat
      wait (wsem);
         WRITEUNIT;
      signal (wsem)
   forever
end;
begin
   readcount := 0;
   parbegin
      reader;
      writer
   parend
end.
```

Figure 5.28 A Solution to the Readers/Writers Problem Using Semaphores: Readers have priority

```
program readersandwriters;
var readcount, writecount: integer;
        x, y, z, wsem, rsem: semaphore (:= 1);
procedure reader;
begin
    repeat
        wait (z);
            wait (rsem);
                wait (x);
                    readcount := readcount + 1;
                    if readcount = 1 then wait (wsem);
                signal (x);
            signal (rsem);
        signal (z);
        READUNIT;
        wait (x);
            readcount := readcount - 1;
            if readcount = 0 then signal (wsem);
        signal (x)
    forever
end;
procedure writer;
begin
    repeat
        wait (y);
            writecount := writecount + 1;
            if writecount = 1 then wait (rsem);
        signal (y);
        wait (wsem);
        WRITEUNIT;
        signal (wsem);
        wait (y);
            writecount := writecount - 1;
            if writecount = 0 then signal (rsem);
        signal (y)
    forever
end;
begin
    readcount, writecount := 0;
    parbegin
        reader;
        writer
    parend
end.
```

Figure 5. 29 A Solution to the Readers/Writers Problem Using Semaphores: Writers have priority

```
program        mutualexclusion;
const n = . . . ; (*number of processes*);
procedure P(i: integer);
var msg: message;
begin
    repeat
        receive (mutex, msg);
        < critical section >;
        send (mutex, msg);
        < remainder >
    forever
end;
begin (* main program *)
    create_mailbox (mutex);
    send (mutex, null);
    parbegin
        P(1);
        P(2);
        . . .
        P(n)
    parend
end.
```

Figure 5.26 Mutual Exclusion Using Messages

```
program   producerconsumer
    monitor    boundedbuffer;
            buffer: array [0 . . N] of  char;          {space for N items}
            nextin, nextout: integer;                  {buffer pointers}
            count: integer;                            {number of items in buffer}
            notfull, notempty: condition;              {for synchronization}

    procedure append (x: char);
            begin
                if count = N then cwait(notfull);      {buffer is full; avoid overflow}
                buffer[nextin] := x;
                nextin := nextin + 1 mod N;
                count := count + 1;                    {one more item in buffer}
                csignal(notempty);                     {resume any waiting consumer}
            end;
    procedure take (x: char);
            begin
                if count = 0 then cwait(notempty);     {buffer is empty; avoid underflow}
                x := buffer[nextout];
                nextout := nextout + 1 mod N;
                count := count - 1;                    {one fewer item in buffer}
                csignal(notfull);                      {resume any waiting producer}
            end;
    begin                                              {monitor body}
            nextin := 0; nextout := 0; count := 0      {buffer initially empty}
    end;

procedure producer;
var         x: char;
begin
    repeat
        produce(x);
        append(x);
    forever
end;
procedure consumer;
var         x: char;
begin
    repeat
        take(x);
        consume(x);
    forever
end;
begin (*main program*)
    parbegin
        producer; consumer
    parend
end.
```

Figure 5.23 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```
program   barbershop1;
var       max_capacity: semaphore (:= 20);
          sofa: semaphore (:= 4);
          barber_chair, coord: semaphore (:= 3);
          cust_ready, finished, leave_b_chair, payment, receipt: semaphore (:= 0);

procedure customer;
begin
    wait(max_capacity);
    enter shop;
    wait(sofa);
    sit on sofa;
    wait(barber_chair);
    get up from sofa;
    signal(sofa);
    sit in barber chair;
    signal(cust_ready);
    wait(finished);
    leave barber chair;
    signal(leave_b_chair);
    pay;
    signal(payment);
    wait(receipt);
    exit shop;
    signal(max_capacity)
end;

procedure barber;
begin
    repeat
        wait(cust_ready);
        wait(coord);
        cut hair;
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    forever
end;

procedure cashier;
begin
    repeat
        wait(payment);
        wait(coord);
        accept pay;
        signal(coord);
        signal(receipt);
    forever
end;

begin (*main program*)
    parbegin
        customer; . . . 50 times; . . . customer;
        barber; barber; barber;
        cashier
    parend
end.
```

Figure 5.20 An Unfair Barbershop

```
program  barbershop2;
var      max_capacity: semaphore (:= 20);
         sofa: semaphore (:= 4);
         barber_chair, coord: semaphore (:= 3);
         mutex1, mutex2: semaphore (:=1);
         cust_ready, leave_b_chair, payment, receipt: semaphore (:= 0);
         finished: array[1..50] of semaphore (:=0);
         count: integer;

procedure customer;
var  custnr:  integer;
begin
    wait(max_capacity);
    enter shop;
    wait(mutex1);
    count := count + 1;
    custnr := count;
    signal(mutex1);
    wait(sofa);
    sit on sofa;
    wait(barber_chair);
    get up from sofa;
    signal(sofa);
    sit in barber chair;
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave barber chair;
    signal(leave_b_chair);
    pay;
    signal(payment);
    wait(receipt);
    exit shop;
    signal(max_capacity)
end;

procedure barber;
var  b_cust:  integer;
begin
    repeat
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut hair;
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    forever
end;

procedure cashier;
begin
    repeat
        wait(payment);
        wait(coord);
        accept pay;
        signal(coord);
        signal(receipt);
    forever
end;

begin (*main program*)
    count := 0;
    parbegin
        customer; . . . 50 times; . . . customer;
        barber; barber; barber;
        cashier
    parend
end.
```

Figure 5.21 A Fair Barbershop