```
/* program mutualexclusion */
const int n = /* number of processes */ ;

void P(int i)
{
    while (true)
    {
        entercritical (i);
        /* critical section */;
        exitcritical (i);
        /* remainder */;
    }
}
void main( )
{
    parbegin (P(R1), P(R2), . . ., P(Rn));
}
```

Figure 5.1 Mutual Exclusion

```
boolean flag [2];
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        while (flag [1])
                if (turn == 1)
                {
                    flag [0] = false;
                        while (turn == 1)
                                /* do nothing */;
                    flag [0] = true;
                }
        /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
void P1( )
{
    while (true)
    {
        flag [1] = true;
        while (flag [0])
            if (turn == 0)
            {
                flag [1] = false;
                while (turn == 0)
                    /* do nothing */;
                flag [1] = true;
            }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Figure 5.3 Dekker's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
     while (true)
     {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1)
                    /* do nothing */;
          /* critical section   */;
          flag [0] = false;
          /* remainder   */;
     }
}
void P1()
{
     while (true)
     {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0)
                    /* do nothing */;
          /* critical section   */;
          flag [1] = false;
          /* remainder   */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Figure 5.4 Peterson's Algorithm for Two Processes

```
struct semaphore {
        int count;
        queueType queue;
}

void wait(semaphore s)
{
        s.count--;
        if (s.count < 0)
        {
            place this process in s.queue;
            block this process
        }
}
void signal(semaphore s)
{
        s.count++;
        if (s.count <= 0)
        {
            remove a process P from s.queue;
            place process P on ready list;
        }
}
```

Figure 5.6 A Definition of Semaphore Primitives

```
struct binary_semaphore {
        enum (zero, one) value;
        queueType queue;
};

void waitB(binary_semaphore s)
{
     if (s.value ==  1)
            s.value = 0;
     else
        {
            place this process in s.queue;
            block this process;
        }
}
void signalB(semaphore s)
{
     if (s.queue.is_empty())
            s.value = 1;
     else
     {
            remove a process P from s.queue;
            place process P on ready list;
     }

}
```

Figure 5.7 A Definition of Binary Semaphore Primitives

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        wait(s);
        /* critical section */;
        signal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.9 Mutual Exclusion Using Semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
    while (true)
    {
        produce();
        waitB(s);
        append();
        n++;
        if (n==1)
                signalB(delay);
        signalB(s);
    }
}
void consumer()
{
    waitB(delay);
    while (true)
    {
        waitB(s);
        take();
        n--;
        signalB(s);
        consume();
        if (n==0)
                waitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
    while (true)
    {
        produce();
        waitB(s);
        append();
        n++;
        if (n==1) signalB(delay);
        signalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    waitB(delay);
    while (true)
    {
        waitB(s);
        take();
        n--;
        m = n;
        signalB(s);
        consume();
        if (m==0) waitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5.13 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
    while (true)
    {
        produce();
        wait(s);
        append();
        signal(s);
        signal(n);
    }
}
void consumer()
{
    while (true)
    {
        wait(n);
        wait(s);
        take();
        signal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

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

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
    while (true)
    {
        produce();
        wait(e);
        wait(s);
        append();
        signal(s);
        signal(n)
    }
}
void consumer()
{
    while (true)
    {
        wait(n);
        wait(s);
        take();
        signal(s);
        signal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;

void customer ()
{
    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)
}

void barber()
{
    while (true)
    {
        wait(cust_ready);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {   wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

void main()
{
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,         cashier);
}
```

Figure 5.19 An Unfair Barbershop

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;
```

```
void customer()
{
    int custnr;
    wait(max_capacity);
    enter_shop();
    wait(mutex1);
    count++;
    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)
}
```

```
void barber()
{
    int b_cust;
    while (true)
    {
        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);
    }
}
```

```
void cashier()
{
    while (true)
    {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}
```

```
void main()
{   count := 0;
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,
        cashier);
}
```

Figure 5.20   A Fair Barbershop

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                   /* space for N items */
int nextin, nextout;                               /* buffer pointers */
int count;                                         /* number of items in buffer */
int notfull, notempty;                             /* for synchronization */

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

void producer()
char x;
{
    while (true)
    {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true)
    {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 5.22 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```
void append (char x)
{
    while(count == N)
        cwait(notfull);              /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                         /* one more item in buffer */
    cnotify(notempty);               /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0)
    cwait(notempty);                 /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                         /* one fewer item in buffer */
    cnotify(notfull);                /* notify any waiting producer */
}
```

Figure 5.23 Bounded Buffer Monitor Code

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.26 Mutual Exclusion Using Messages

```
const int
    capacity =   /* buffering capacity */ ;
    null =     /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

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

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
        wait (x);
        readcount++;
        if (readcount == 1)
            wait (wsem);
        signal (x);
        READUNIT();
        wait (x);
        readcount--;
        if (readcount == 0)
            signal (wsem);
        signal (x);
    }
}
void writer()
{
    while (true)
    {
        wait (wsem);
        WRITEUNIT();
        signal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Figure 5.28 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
        wait (z);
        wait (rsem);
        wait (x);
        readcount++;
        if (readcount == 1)
        {
            wait (wsem);
        }
        signal (x);
        signal (rsem);
        signal (z);
        READUNIT();
        wait (x);
        readcount--;
        if (readcount == 0)
            signal (wsem);
        signal (x);
    }
}
void writer ()
{
    while (true)
    {
        wait (y);
        writecount++;
        if (writecount == 1)
            wait (rsem);
        signal (y);
        wait (wsem);
        WRITEUNIT();
        signal (wsem);
        wait (y);
        writecount--;
        if (writecount == 0)
            signal (rsem);
        signal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5. 29 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority