Chapter 5 Lecture

Stallings 9ed

Concurrency

Table 5.1 Some Key Terms Related to Concurrency

critical section	A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Concurrency - Definition

“The fact of two or more events happening at the same time”

CS Implications:

- In a single processor system, can two processes truly execute concurrently?
    - Appearance of concurrency, but in actuality only one process can execute
- Scheduler determines which process is executing
- Two “concurrent processes” may execute in arbitrary order, arbitrary interleaving.
- Result - never assume a particular order of execution of two concurrent processes

Difficulties of Concurrency

- Sharing of global resources
- Operating system managing the allocation of resources optimally
- Difficult to locate programming errors

A Simple Example

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

A Simple Example

- Assume
  - single processor
  - 2 processes execute echo
  - global variables

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

- What are the possible outputs?

A Simple Example

Now assume 2 processors

Process P1	Process P2
`.`	`.`
`chin = getchar();`	`    .`
`.`	`chin = getchar();`
`chout = chin;`	`    chout = chin;`
`putchar(chout);`	`.`
`.`	`putchar(chout);`
`.`	`.`

Difficulties of Concurrency

- Sharing of global resources
- Operating system managing the allocation of resources optimally
- Difficult to locate programming errors

When is Concurrency Important?

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

Concurrency

- Multiple applications
  – Multiprogramming
- Structured application
  – Application can be a set of concurrent processes
- Operating-system structure
  – Operating system is a set of processes or threads

Process Interaction

- Processes unaware of each other
- Processes indirectly aware of each other
- Process directly aware of each other

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Competition Among Processes for Resources

- Mutual Exclusion
  - Critical sections
    - Only one program at a time is allowed in its critical section
    - Example only one process at a time is allowed to send command to the printer
- Deadlock
- Starvation

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion cont.

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - In general: A process runs until it invokes an operating system service or until it is interrupted
  - Uni-processor: Disabling interrupts guarantees mutual exclusion
    - Processor is limited in its ability to interleave programs
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void exchange(int register,
              int memory) {
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Mutual Exclusion

- parbegin: initiate all processes and resume program after all Pi’s have terminated

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
           /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

(a) Test and set instruction

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
             exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical section and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical section (which will not be returned).

Software Solutions – Bakery Algorithm

- Also called Lamport’ s bakery algorithm
  - after Leslie Lamport
  - A New Solution of Dijkstra's Concurrent Programming Problem Communications of the ACM 17, 8 (August 1974), 453-455.
- This is a mutual exclusion algorithm to prevent concurrent threads from entering critical sections concurrently
- source: wikipedia

Bakery Algorithm

- Analogy
  - bakery with a numbering machine
  - each customer receives unique number
    - numbers increase by one as customers enter
  - global counter displays number of customer being served currently
    - all others wait in queue
  - after baker is done serving customer the next number is displayed
  - served customer leaves

Bakery Algorithm

- threads and bakery analogy
  - when thread wants to enter critical section it has to make sure it has the smallest number.
    - however, with threads it may not be true that only one thread gets the same number
      - e.g., if number operation is non-atomic
    - if more that one thread has the smallest number then the thread with lowest id can enter
    - use pair (number, ID)
      - In this context (a,b) < (c,d) is equivalent to
      - (a<c) or ((a==c) and (b<d))

Bakery Algorithm

from wikipedia

```
// declaration and initial values of global variables
Entering: array [1..N] of bool = {false};
Number: array [1..N] of integer = {0};

lock(integer i)
{
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[N]);
    Entering[i] = false;
    for (j = 1; j <= N; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
            /* nothing */
        }
    }
}
unlock(integer i) { Number[i] = 0; }

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Peterson's Algorithm 1981

- solves critical section problem
- based on shared memory for communication

```
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 5.3 Peterson's Algorithm for Two Processes

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - ***Wait*** operation decrements the semaphore value
  - ***Signal*** operation increments semaphore value

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

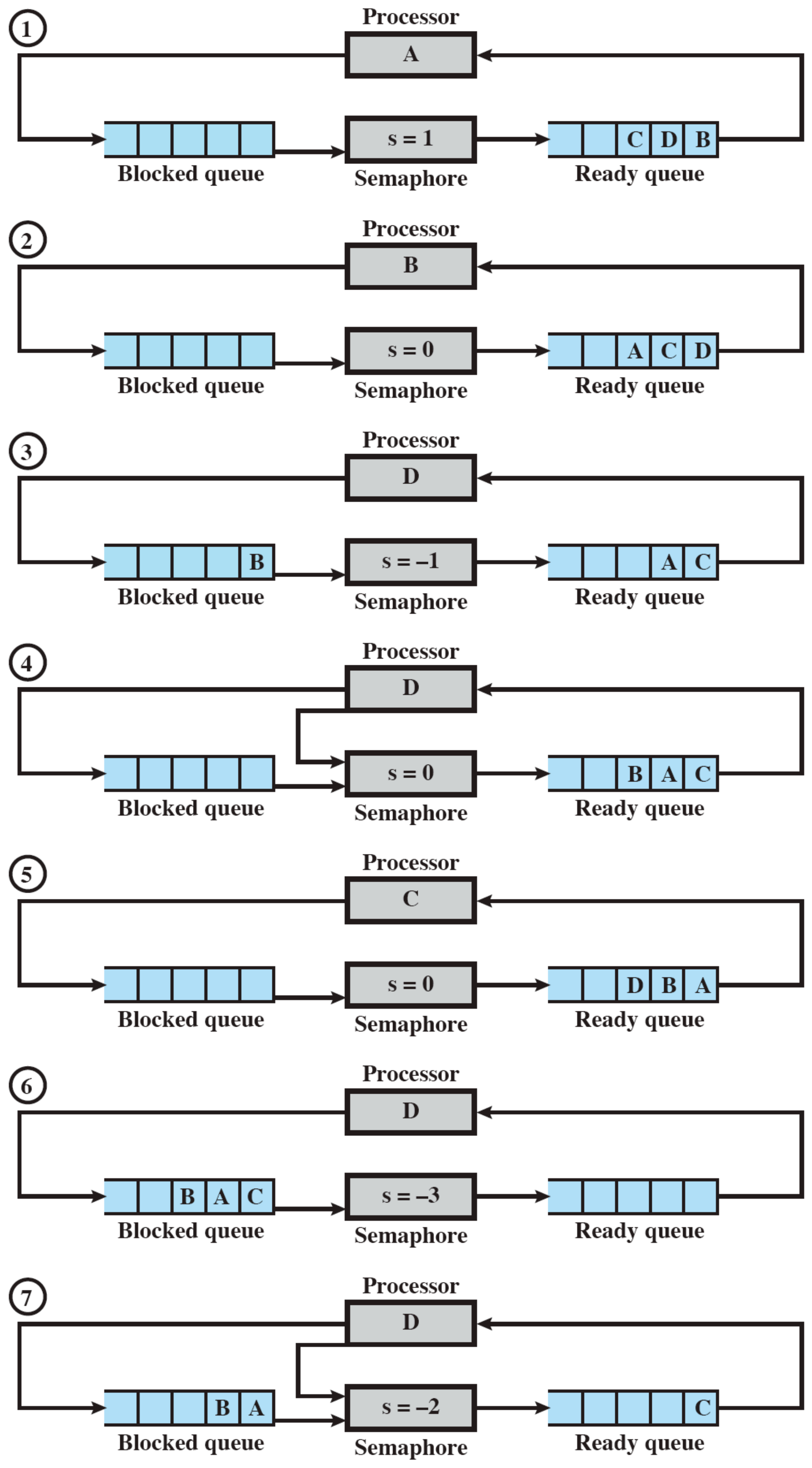
Figure 5.6 A Definition of Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
     }
}
```

Figure 5.7 A Definition of Binary Semaphore Primitives

Assume process A,B and C depend on result of process D

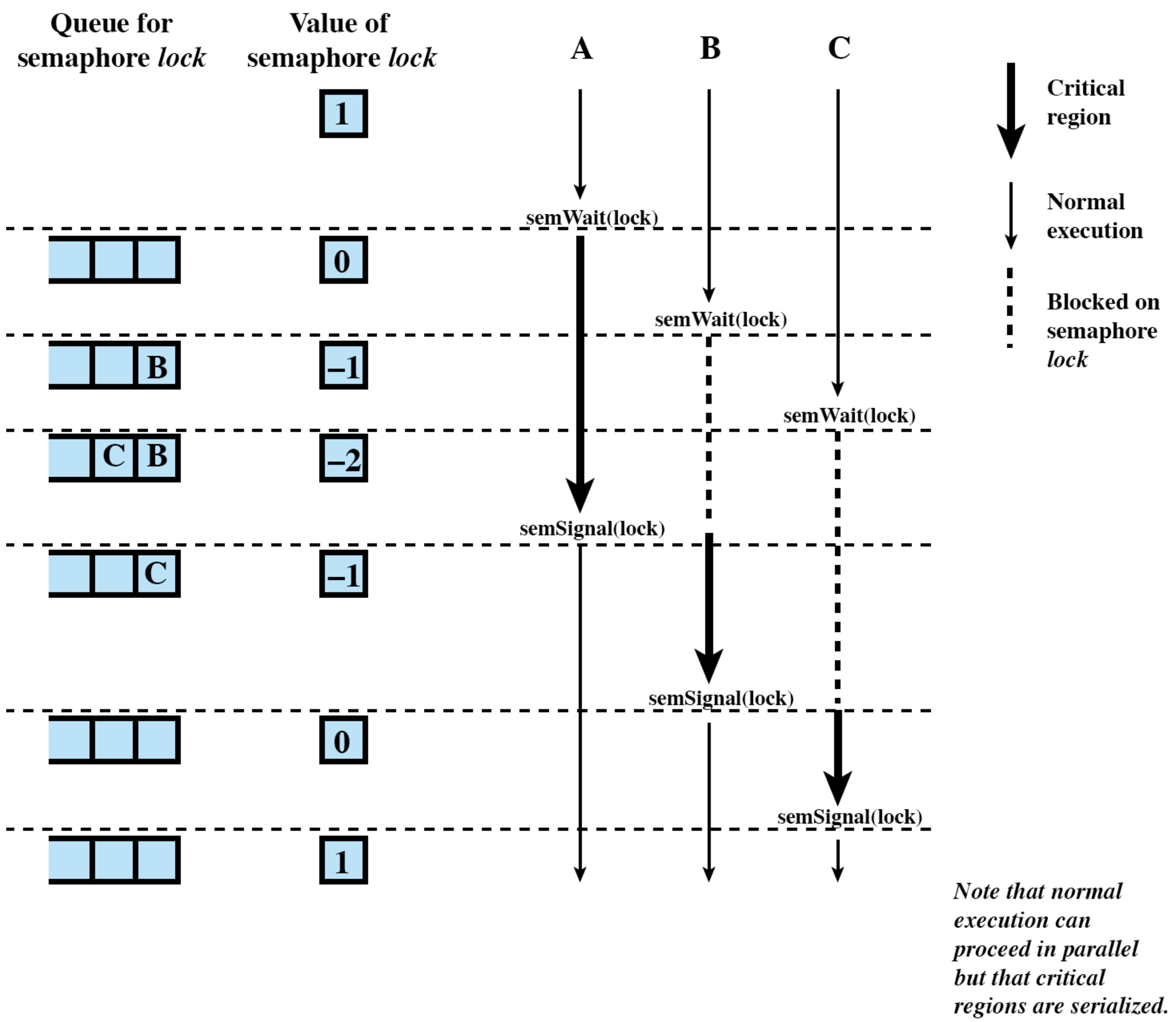
Initially one result of D is available (s = 1)



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

Figure 5.9 Mutual Exclusion Using Semaphores

Assume 3 processes, A, B and C



Producer/Consumer Problem

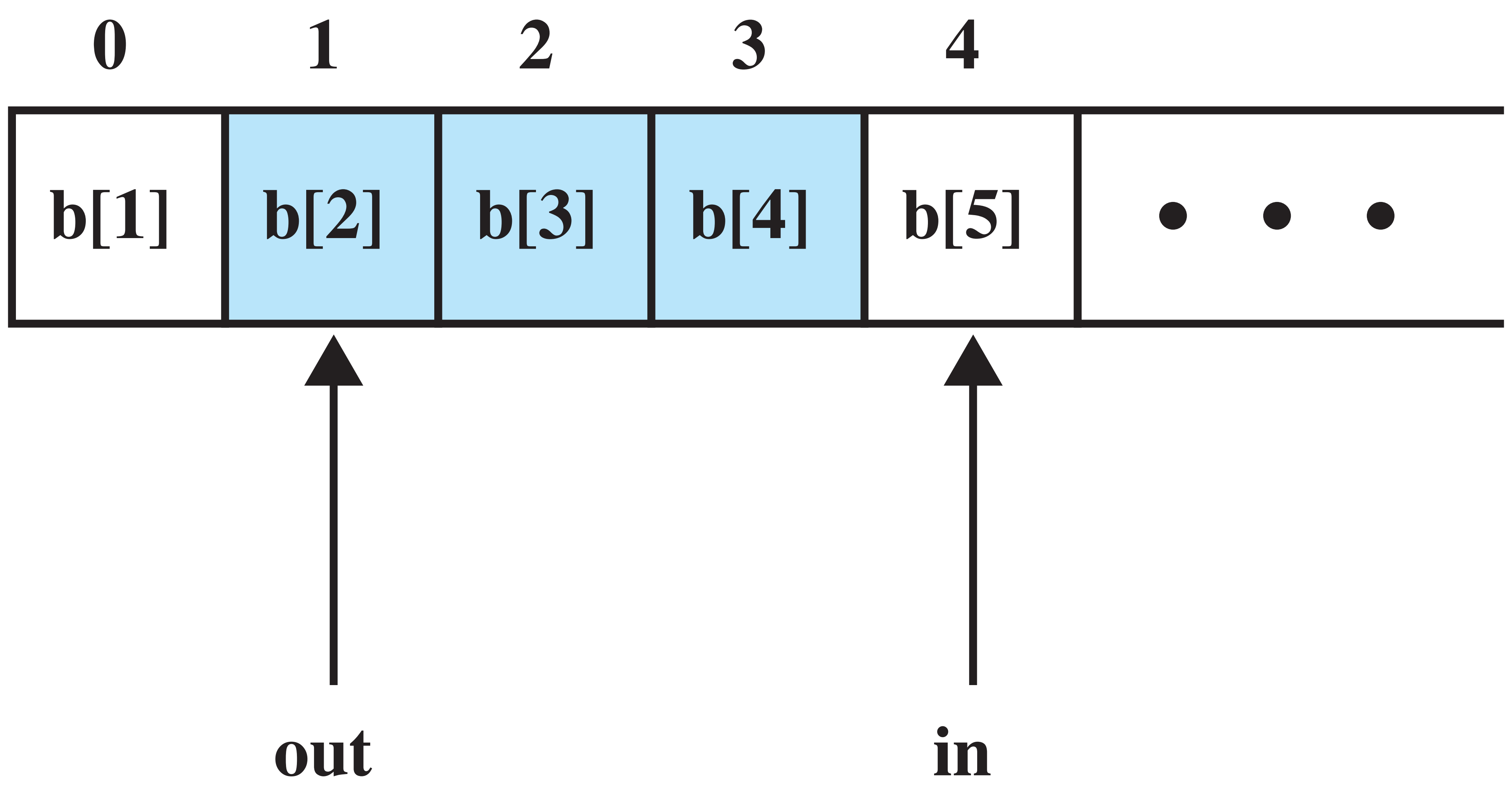
- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

Consumer

```
consumer:
while (true) {
  while (in <= out)
     /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)
     /* do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
     /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

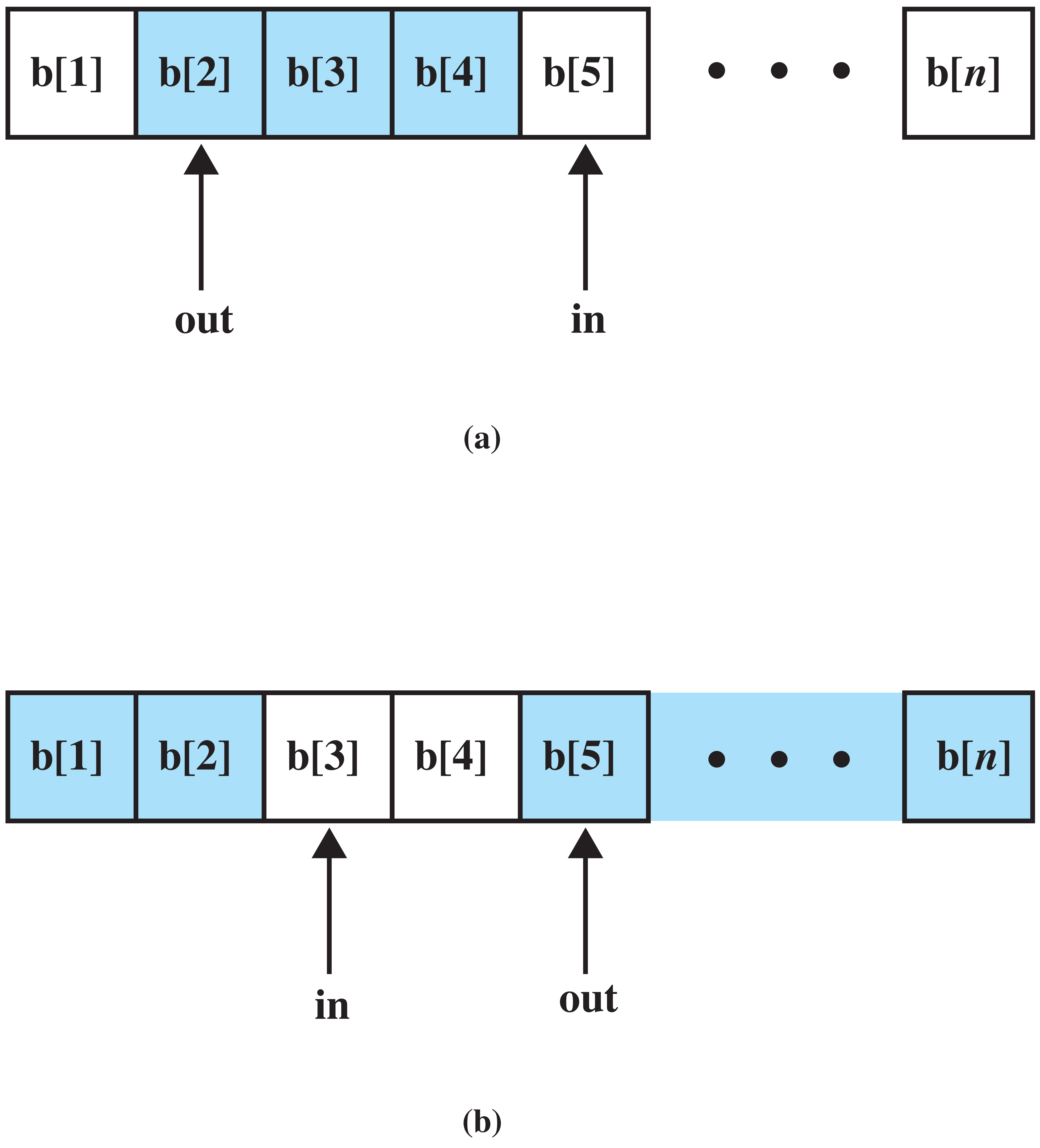


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

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

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

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
       /* do nothing */;
   s.count--;
   if (s.count < 0) {
       /* place this process in s.queue*/;
       /* block this process (must also set s.flag to 0)
*/;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
         /* do nothing */;
   s.count++;
   if (s.count <= 0) {
       /* remove a process P from s.queue */;
       /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
       allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Figure 5.17 Two Possible Implementations of Semaphores

Using Semaphores

- It is difficult to use semaphores
  - see example in Fig 5.12
  - semaphores may be scattered throughout the program
    - difficult to assess overall effect
- Monitors provide similar functionality
  - but are easier to control
  - implemented in languages like Concurrent Pascal, Pascal-Plus, Modula-2 & 3, and Java

Monitors

- A Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time

Monitors

- Provides mutual exclusion facility
- Shared data structure can be protected by placing it into a monitor
- If the data in a monitor represents some resource, then mutual exclusion is guaranteed for that resource

Monitors

- Synchronization support is needed
  - implemented using special data types called *condition variables*
  - these variables are affected by two functions
    - `cwait(c)`
      - suspend calling process on condition `c`
      - now monitor can be used by other process
    - `csignal(c)`
      - resume blocked process after `cwait` on same condition `c`

Monitors

- So what is the difference between the use of cwait and csignal in monitors and the wait and signal of semaphores?
  - Hint: remember what got us in trouble when using semaphores

Monitors

- Monitor wait and signal operations are different from their counterparts in semaphores
  - If a process in a monitor signals and corresponding queue is empty then signal is lost

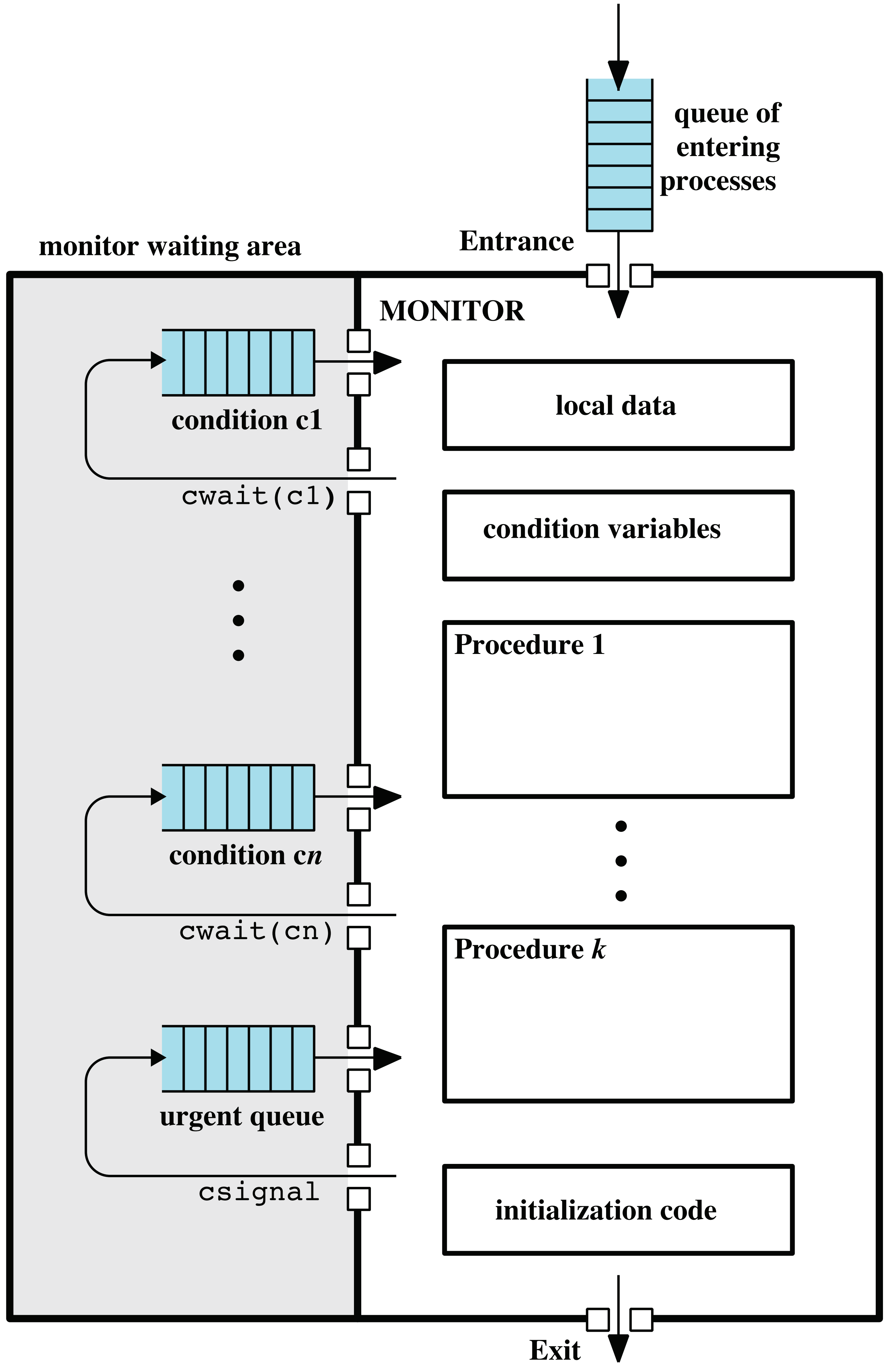


Figure 5.18 Structure of a Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                          /* space for N items */
int nextin, nextout;                                        /* buffer pointers */
int count;                                         /* number of items in buffer */
cond notfull, notempty;            /* condition variables 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 5.19 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 5.20 Bounded Buffer Monitor Code for Mesa Monitor

Message Passing

- Interaction between processes
  - synchronization
  - communication

- One solution to this is message passing
  - works in both tightly and loosely coupled systems

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

Synchronization

- Sender and receiver may or may not be blocking (waiting for message)

- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - This is called a *rendezvous*

Synchronization

- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives

- Nonblocking send, nonblocking receive
  - Neither party is required to wait

Addressing

- Direct addressing
  - Send primitive includes a specific identifier of the destination process
  - Receive primitive could know ahead of time which process a message is expecting
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

Addressing

- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called *mailboxes*
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox
  - relationship between sender & receiver
    - 1-to-1, many-to-1, 1-to-many, many-to-many

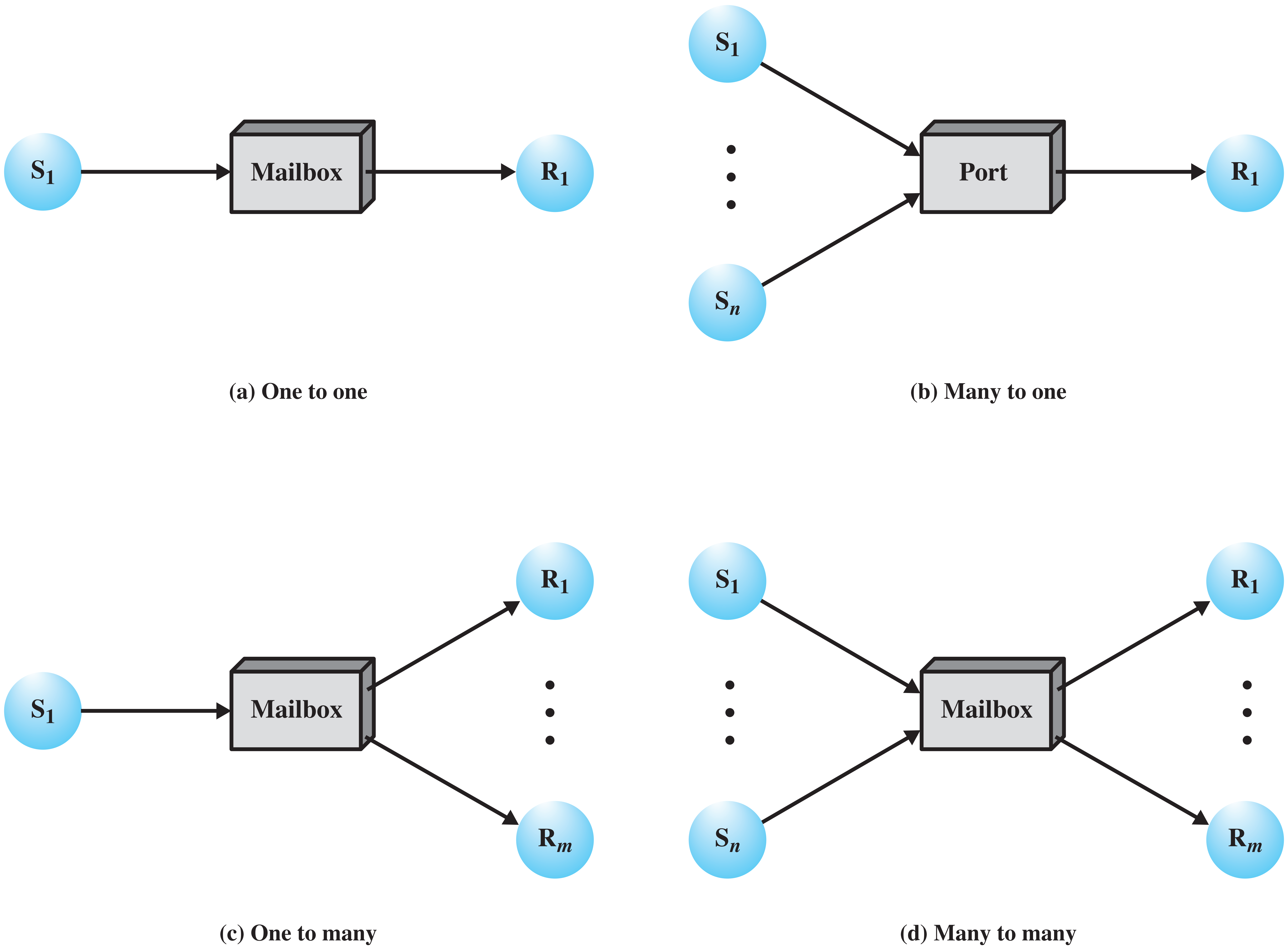


Figure 5.21 Indirect Process Communication

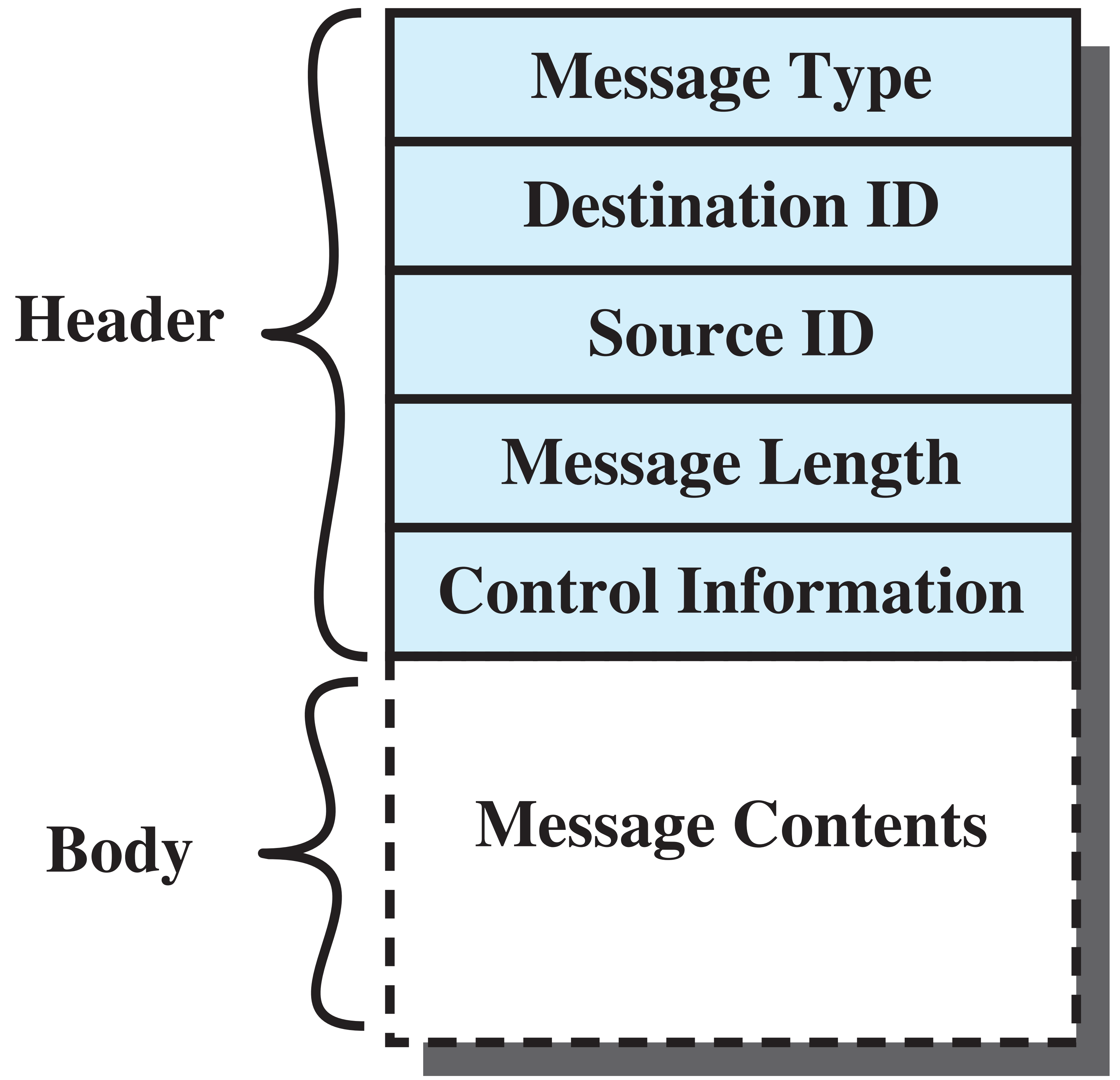


Figure 5.22 General Message Format

Assumptions: non-blocking send, blocking receive

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

Figure 5.23 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 5.24 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Readers/Writers Problem

- Different variations on the theme, e.g.,
  – dedicated readers and dedicated writers
  – they all can read and write
- Here we look at the "dedicated" case
  – Any number of readers may simultaneously read the file
  – Only one writer at a time may write to the file
  – If a writer is writing to the file, no reader may read it

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

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

Figure 5.25 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) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5.26 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Figure 5.27 A Solution to the Readers/Writers Problem Using Message Passing

```
char   rs, sp;
char inbuf[80], outbuf[125] ;
void read()
{
  while (true) {
    READCARD (inbuf);
    for (int i=0; i < 80; i++){
          rs = inbuf [i];
          RESUME squash
    }
    rs = " ";
    RESUME squash;
  }
}
void print()
{
  while (true) {
    for (int j = 0; j < 125; j++){
          outbuf [j] = sp;
          RESUME squash
    }
    OUTPUT (outbuf);
  }
}
```

```
void squash()
{
  while (true) {
    if (rs != "*") {
          sp = rs;
          RESUME print;
    }
    else{
      RESUME read;
      if (rs == "*") {
          sp = "↑";
          RESUME print;
      }
      else {
        sp = "*";
        RESUME print;
        sp = rs;
        RESUME print;
      }
    }
    RESUME read;
  }
}
```

Figure 5.28 An Application of Coroutines