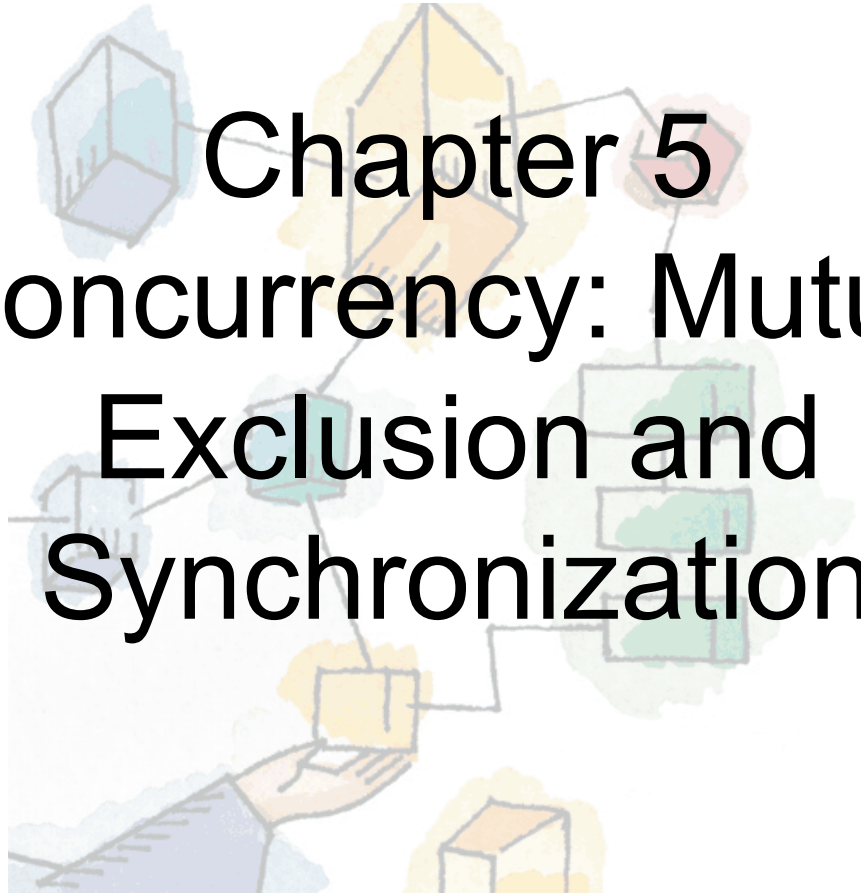
Operating Systems:
Internals and Design Principles, 6/E
William Stallings



Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Patricia Roy
Manatee Community College, Venice, FL

Concurrency

When several processes/threads have access to some shared resources

- Multiple applications
- Structured applications
- Operating system structure
- Multithreaded processes

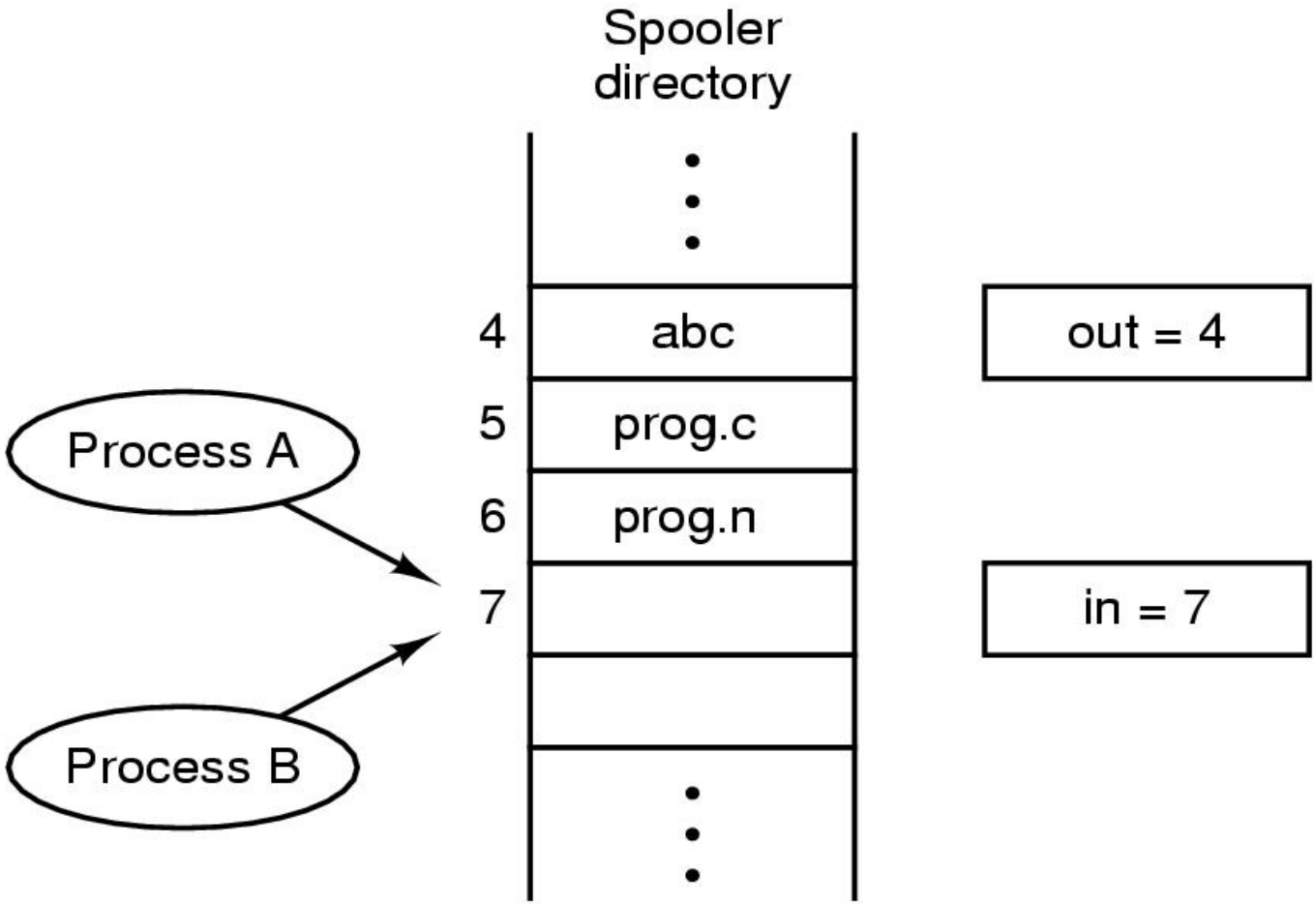
Table 5.1 Some Key Terms Related to Concurrency

atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
critical section	A section of code within a process that requires access to shared resources and that must 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 states 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.

Difficulties of Concurrency

- Sharing of global resources – coordinated access to shared resources
- Operating system managing the allocation of resources optimally – do we know the future behaviour (needed resources) of processes?
- Difficult to locate programming errors

Race Conditions



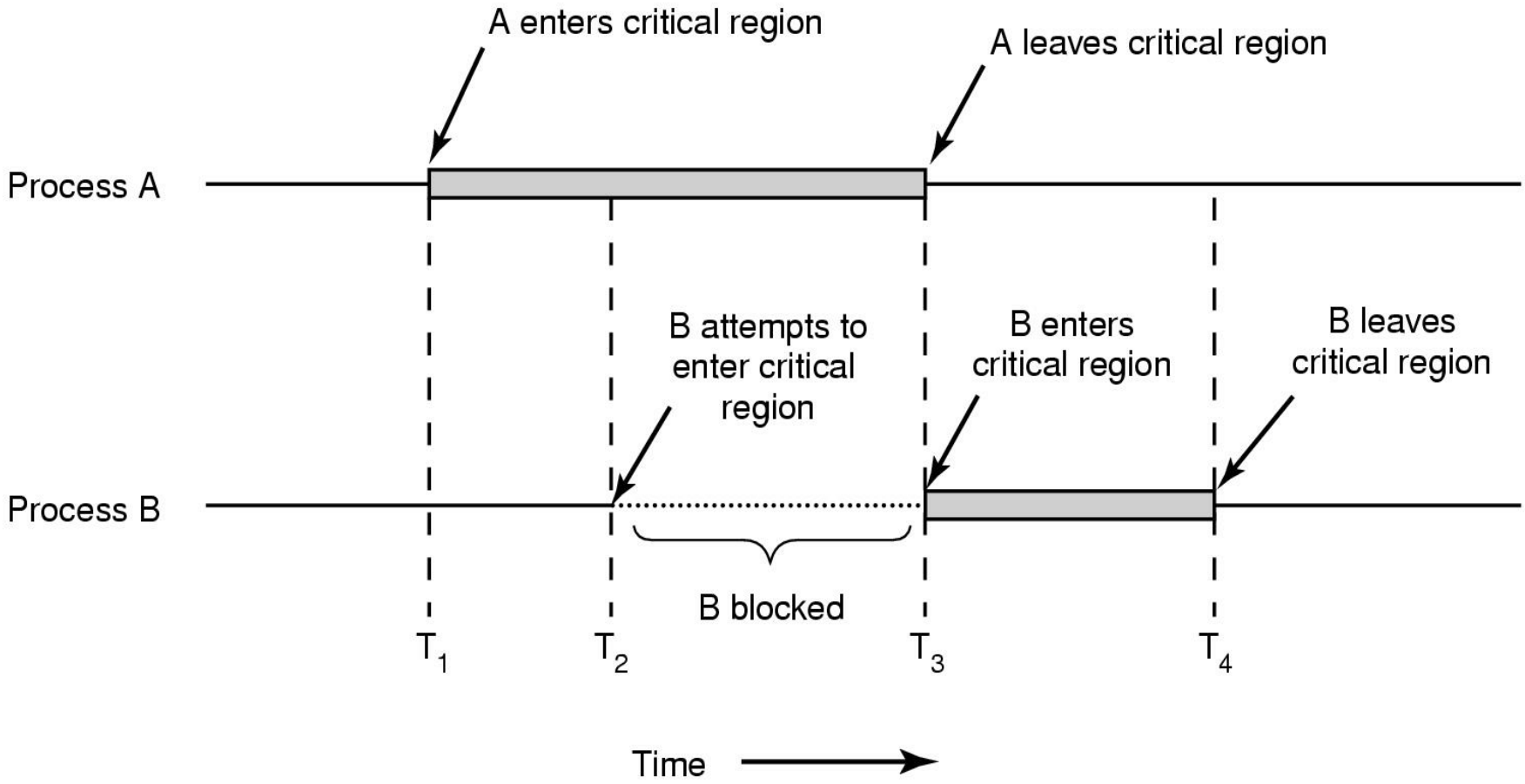
Two processes want to access shared memory at the same time.

The final result depends on who runs precisely when (determined by the scheduler).

Potential Problems

- Data incoherency
- Deadlock: processes are “frozen” because of mutual dependency on each other
- Starvation: some of the processes are unable to make progress (i.e., to execute useful code)

Critical Regions



Mutual exclusion using critical regions

Mutual exclusion

Critical region: part of the program where shared memory is accessed.

Four conditions for correct and efficient communication:

1. Mutual exclusion: No two processes simultaneously in their critical regions
2. No assumptions made about speeds (or numbers) of CPUs
3. Progress: No process running outside its critical region may block another process to enter
4. Fairness, i.e., no starvation: No process must wait forever to enter its critical region (assuming fair scheduling!)

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical region problem

(a) Process 0 (b) Process 1

Invariance: turn=id of process (trying to get) in c.s.

Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
    int other;                            /* number of the other process */

    other = 1 – process;                  /* the opposite of process */
    interested[process] = TRUE;           /* show that you are interested */
    turn = process;                       /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
    interested[process] = FALSE;          /* indicate departure from critical region */
}
```

Peterson's Solution (ctd.)

- `Interested(process)=False =>` process is not in and does not want to enter critical section
- If both are interested, a process can enter only if it is the other’s turn (the other process arrived later)
- Works only for two processes (generalization is possible)
- Works in distributed systems (no special instruction needed)
- Process loops when unable to enter c.s.

Mutual Exclusion: Disabling Interrupts

- A process runs until it invokes an operating system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Processor is limited in its ability to interleave programs
- Will not work in multiprocessor architecture

Should a user process be allowed to disable interrupts?

Mutual Exclusion: Exchange

- Exchange instruction

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

Mutual Exclusion with Exchange

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

(b) Exchange instruction

Mutual Exclusion: Compare&Swap

- Compare&Swap Instruction

```
int compare_and_swap (int word, int testval,
   int newval)
{
   int oldval;
   oldval = word;
   if (oldval == testval) word = newval;
   return oldval;
}
```

Thus: c_a_s(0,0,1)=0 and c_a_s(1,0,1)=1 (and word=1)

Mutual Exclusion with Compare&Swap

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

}
```

(a) Compare and swap instruction

Invariances

Exchange:

process in c.s. => bolt = 1

thus other process cannot enter c.s.

Compare&Swap:

#(proc. in c.s.) = bolt

one process in in c.s. <=> bolt = 1

thus at most one proc. can enter/be in c.s.

Mutual Exclusion Machine-Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It can be used to support multiple critical sections

Mutual Exclusion Machine-Instruction: Disadvantages

- **Busy-waiting** consumes processor time: processes spin on variable
- **Livelock** is possible: process waits on a variable while other process waits on another variable – none of them can release
- **Priority inversion problem**: low priority process in critical section, high priority process wants to enter the critical section

Semaphores

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

Semaphore Operations

- Semaphore is a variable that has an integer value
  - May be initialized to a non-negative number
  - Wait (down, request) operation decrements semaphore value; if value negative, process is blocked
  - Signal (up, release) operation increments semaphore value; one of the blocked processes (if any) is unblocked

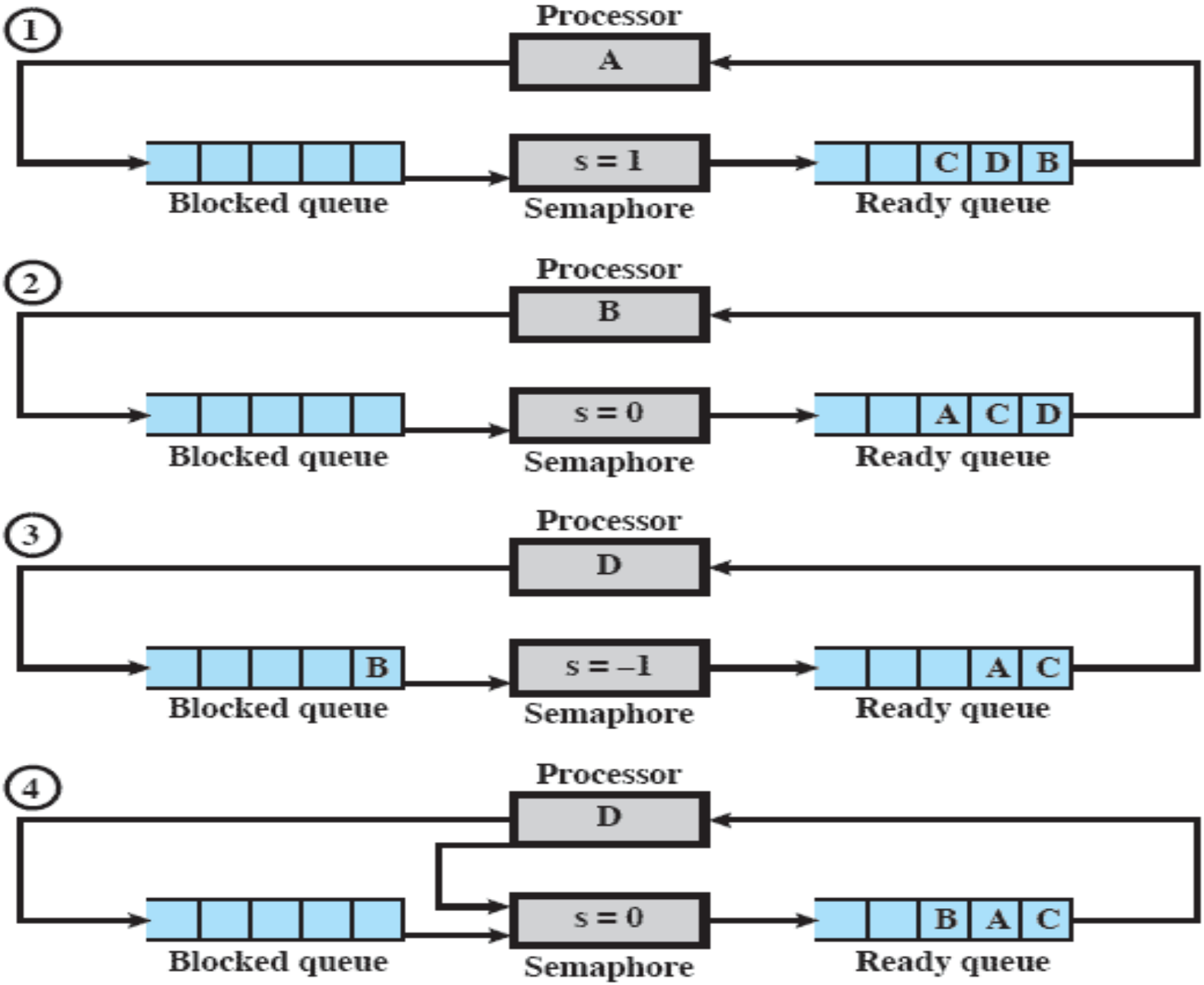
```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
     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

Figure 5.3 A Definition of Semaphore Primitives

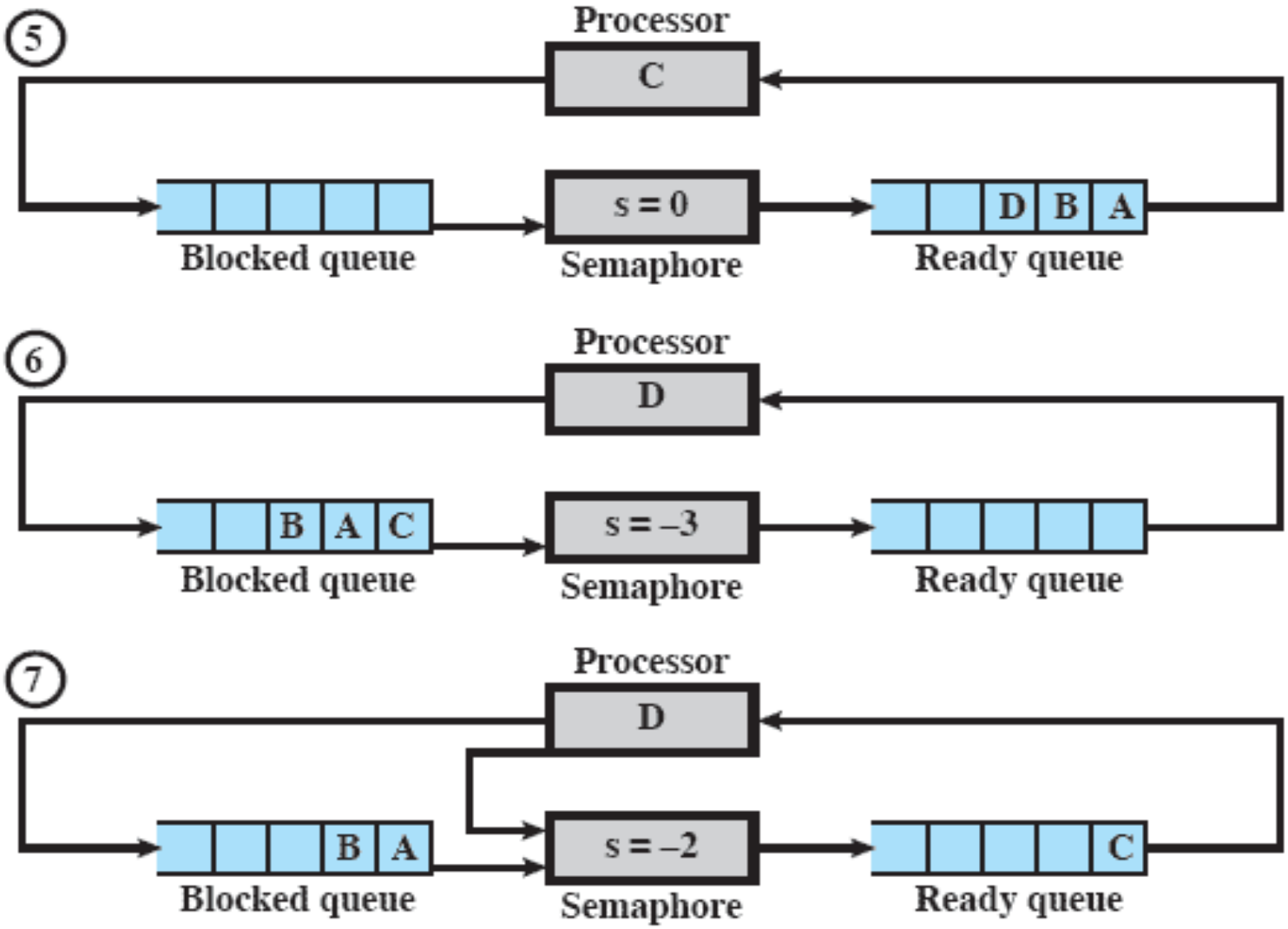
```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
                  /* place this process in s.queue */;
                  /* block this process */;
      }
}
void 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 5.4 A Definition of Binary Semaphore Primitives

Example of Semaphore Mechanism: D performs signal



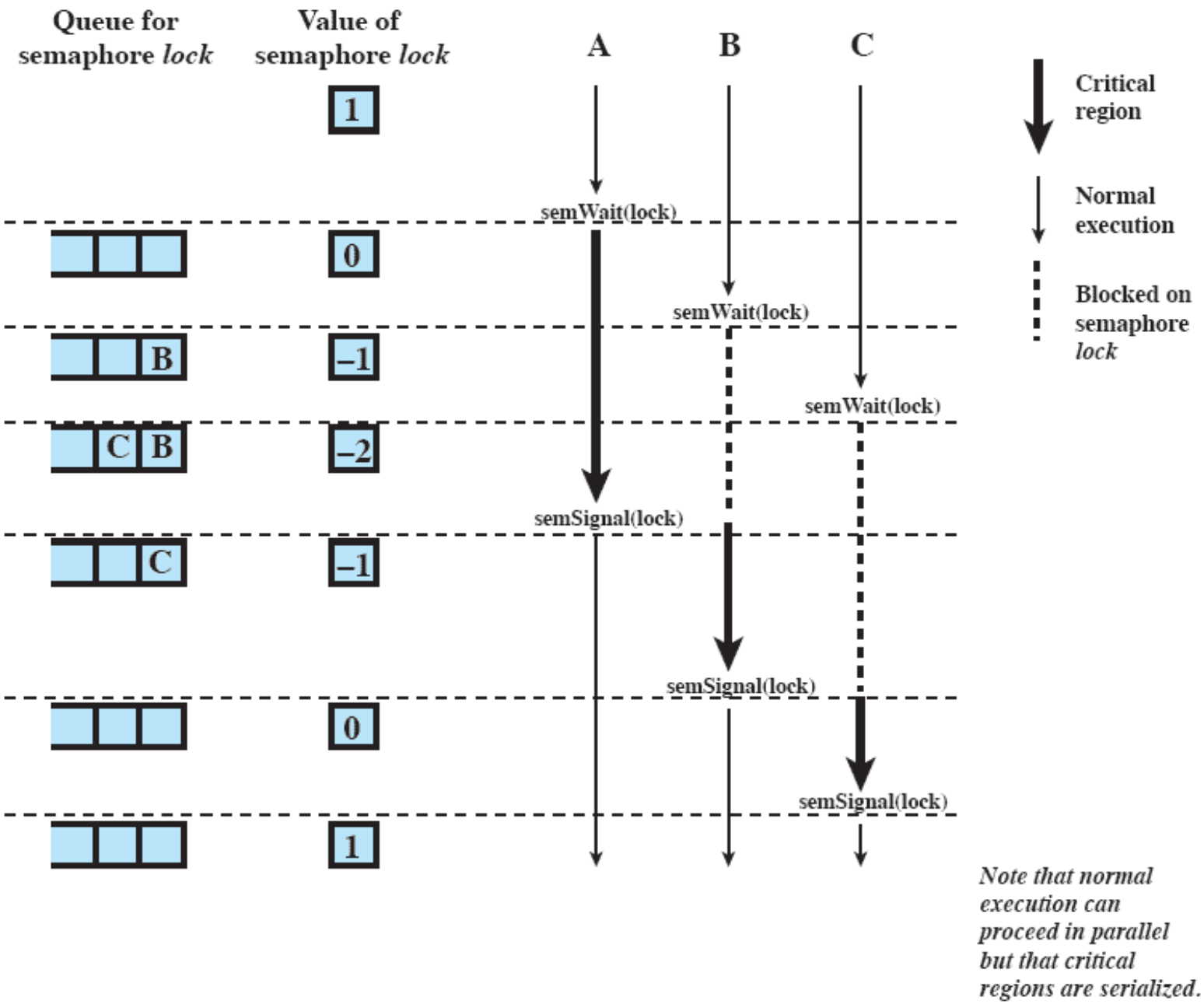
Example of Semaphore Mechanism



```
/* 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));
}
```

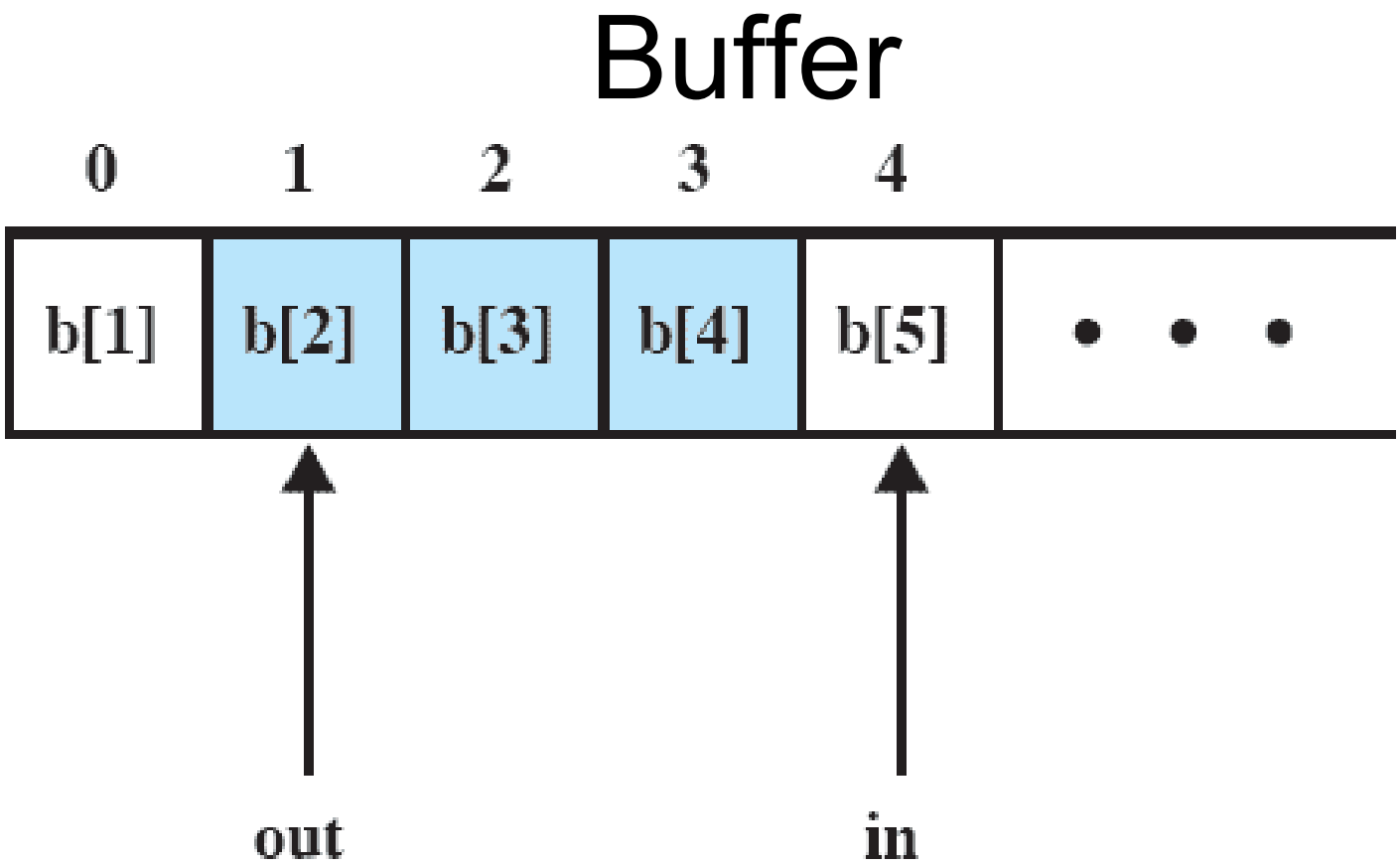
Figure 5.6 Mutual Exclusion Using Semaphores

Processes Using Semaphore



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 can’t add data into full buffer and consumer can’t remove data from empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

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 */
}
```

Incorrect Solution (early signal)

```
/* 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);
}
```

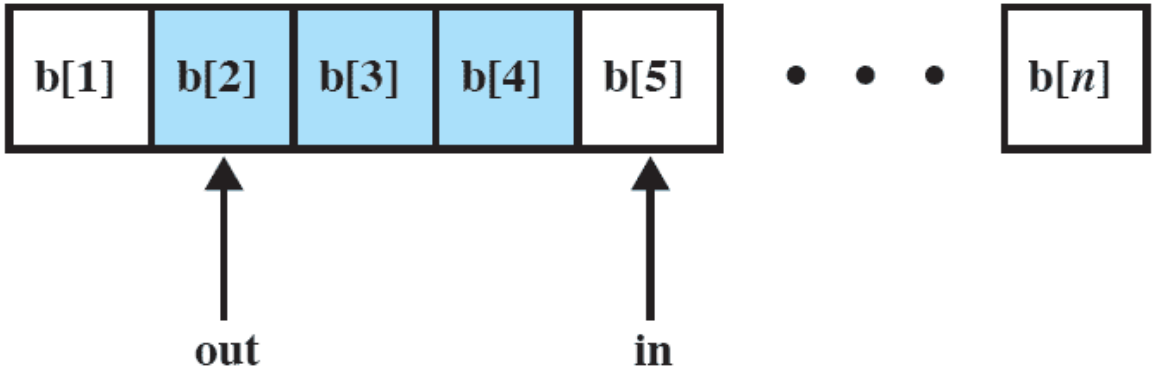
Correct Solution with Binary Semaphores

```
/* 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);
}
```

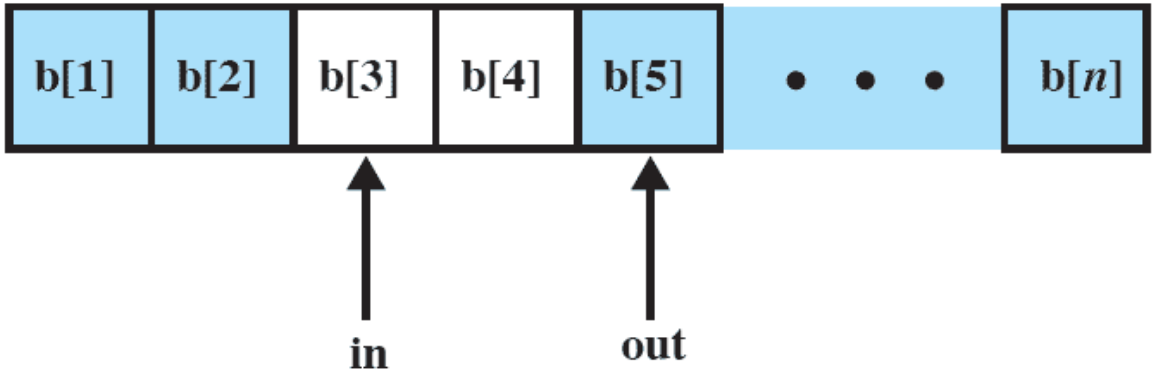
```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
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

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

Circular Buffer



(a)



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 */
}
```

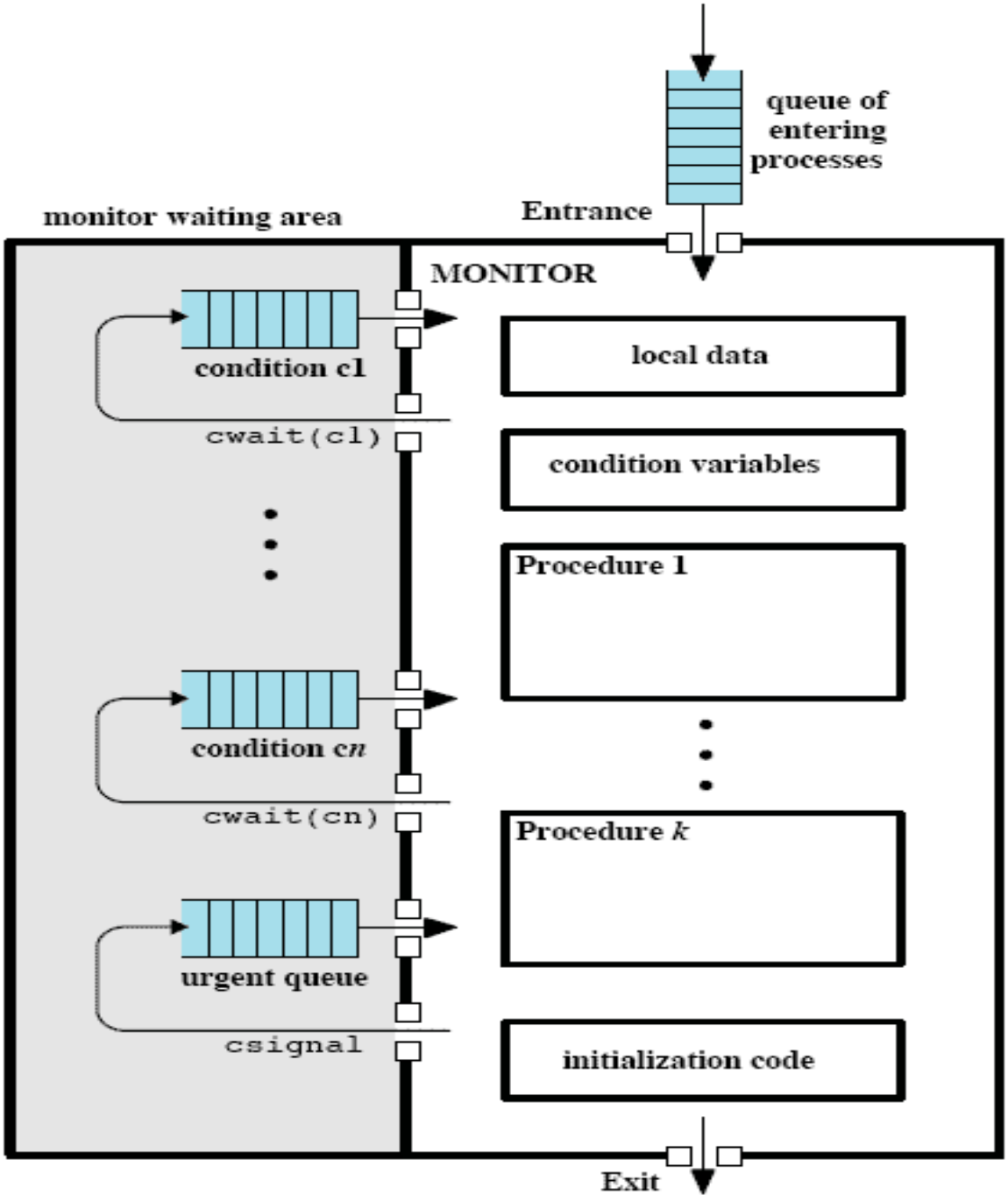
Correct Solution with Bounded Buffer

```
/* 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);
}
```

Monitors

- 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 – mutual exclusion is guaranteed

Structure of a Monitor



Solution Using Monitor

```
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);
}
```

Solution Using Monitor (cont.)

```
/* 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 */
}
```

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
  - Called a rendezvous

Synchronization (2)

- Non-blocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Non-blocking send, non-blocking 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 from which process a message is expected
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

Addressing (2)

- 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

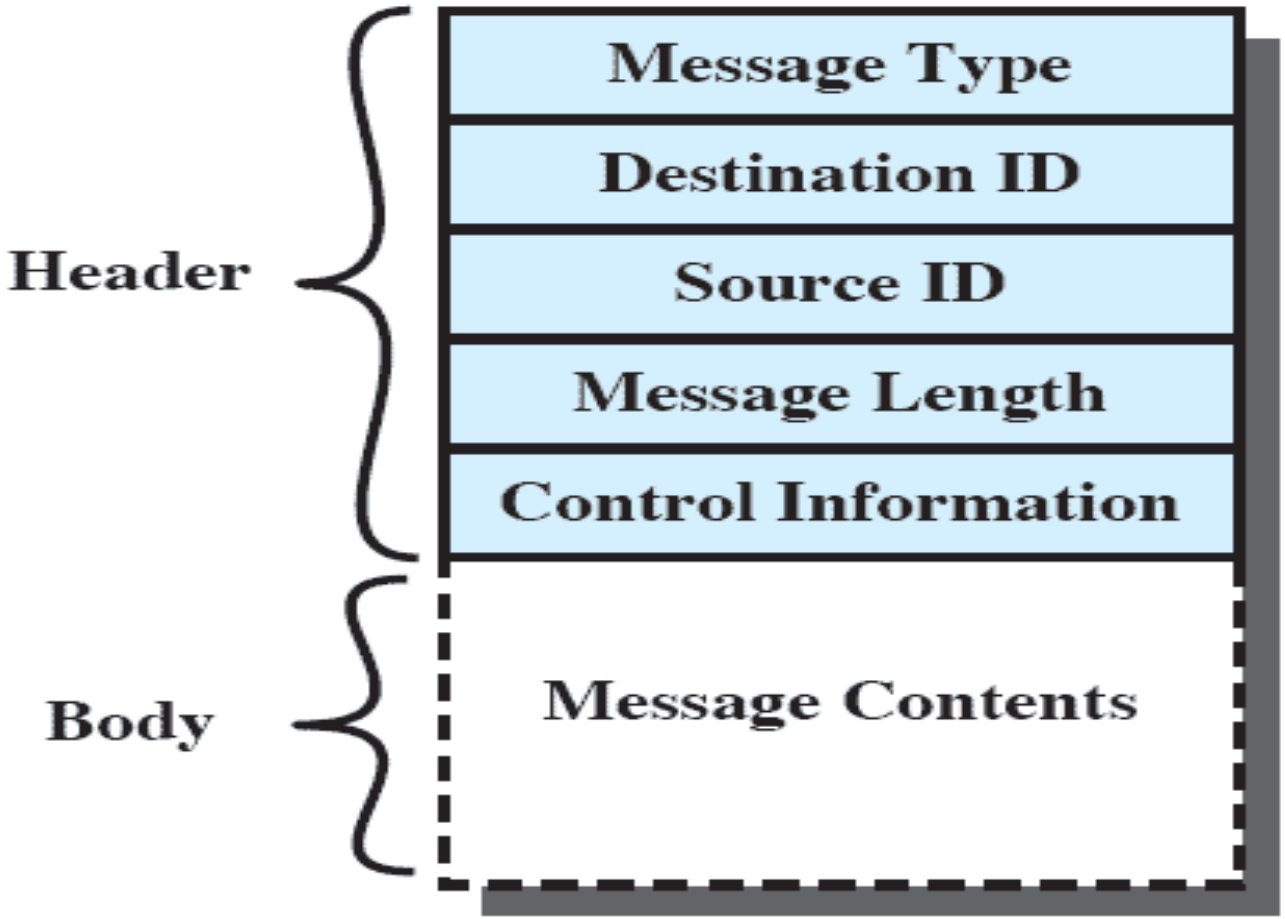


Figure 5.19 General Message Format

```
/* 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 5.20 Mutual Exclusion Using Messages

Producer/Consumer 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);
}
```

Readers/Writers Problem

- 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

Readers Have Priority

```
/* 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);
}
```

Writers 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);
    }
}
```

Writers Have Priority

```
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);
}
```

R-W with Message Passing

```
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++;
         }
      }
}
```