Semaphores

- Two or more processes can cooperate through signals
- A semaphore is a special variable used for signaling
- **semSignal (s) & semWait (s):** primitive used to transmit a signal or to wait for a signal
- 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 value (# of processes that pass without being blocked)
  - **semWait (s)** decrements the semaphore value; if the value becomes < 0, the process is blocked;
  - **semSignal (s)** increments the semaphore value; only if the value remains <= 0 another process in the queue is unblocked.

Semaphore Primitives

```
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 */;
      }
}
```

Binary Semaphores

- Semaphore is a variable that can be initialized to either 0 or 1
  - **semWaitB (s)** checks the value. If it is 0 the process is blocked. If it is 1, it is set to 0 and the process continues;
  - **semSignalB (s)** checks the queue; if it is empty, sets the semaphore to 1. otherwise, puts one of the queue processes in the ready list.

Binary 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 */;
      }
}
```

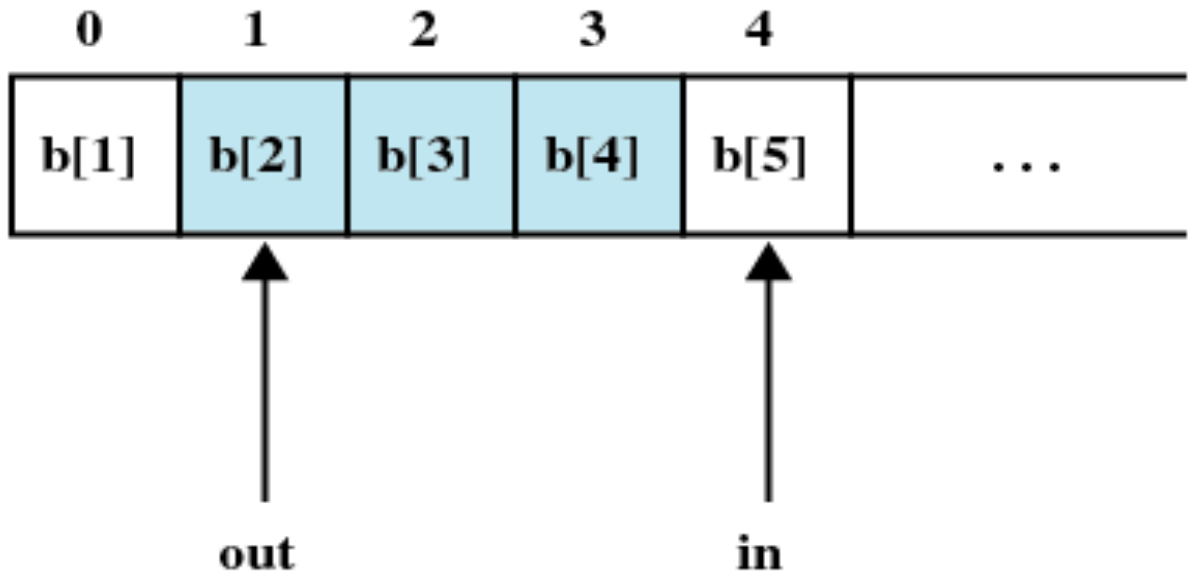
Mutual Exclusion Using Semaphores

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

Producer/Consumer Problem

- 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
- Conditions:
  - Only one producer or consumer may access the buffer at any one time
  - Producers to not write if buffer full
  - Consumer to not read if buffer empty

Producer/Consumer Problem



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Abstract definition of producer & consumer

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

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

First solution attempt:

- Preserve two properties
  - Buffer has elements
  - Is accessed in an exclusive way
- Idea:
  - n keeps track of the items (= in - out)
  - semaphore **delay** is used to block the consumer
  - Semaphore **s** is used for mutual exclusion among all

```
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);
     }
}
```

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

Idea

- Add variable **m** which keeps track of the consumer's view of the situation (value of **n**)

Correct Solution

```
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);
     }
}
```

Solution with General Semaphores

```
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();
     }
}
```

Inverting commands

```
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();
     }
}
```

- Inverting semSignals?

Inverting commands

```
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();
      }
}
```

- Inverting semSignals: **OK**
  - Consumer has to wait for two semaphores anyways
- Inverting semWaits?

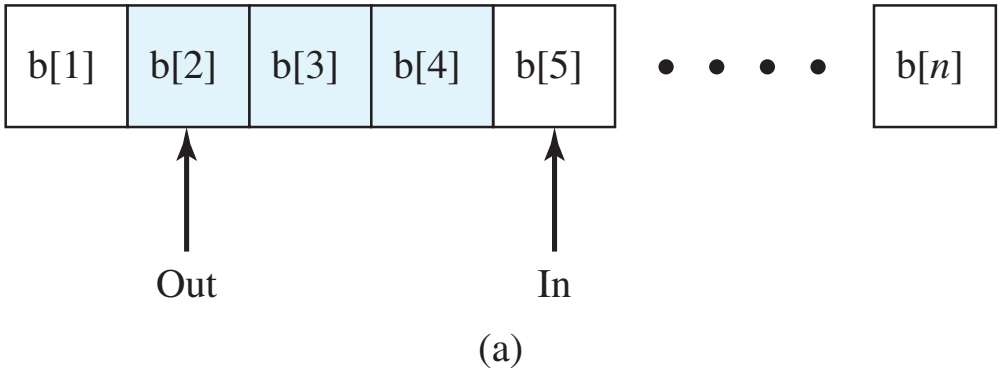
Inverting commands

```
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();
      }
}
```

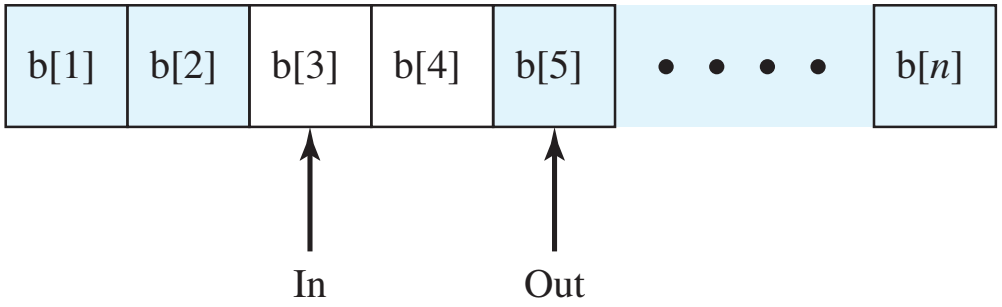
- Inverting semSignals: **OK**
  - Consumer has to wait for two semaphores anyways
- Inverting semWaits: **Deadlock**
  - Consumer gets lock on semaphore s when n == 0
  - Producer cannot write

Finite Circular Buffer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed



(a)



(b)

Abstract behavior of producer/ consumer with circular buffer

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

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

- Idea: keep track of consumed elements

Abstract behavior of producer/ consumer with circular buffer

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

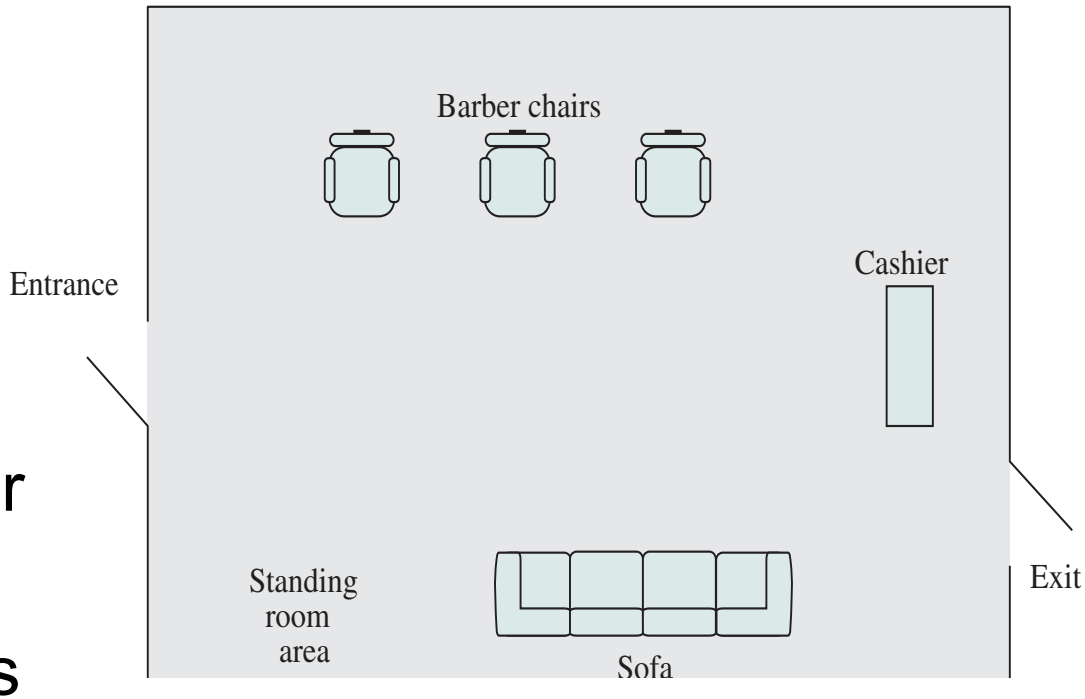
```
consumer:
while (true) {
   while (in == out)
    /* do nothing */;
   w = b[out];
   out = (out + 1) % n;
   /* consume item w */
}
```

- Idea: keep track of consumed elements
  - Sem. **e (empty)** incremented by consumer and decremented by producers

```
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();
     }
}
```

Barbershop Problem

- Three chairs
- Three barbers
- One cashier
- Sofa for four waiting customers
- Standing room for other customers
- Capacity: 20 customers



Barbershop Problem: semaphores

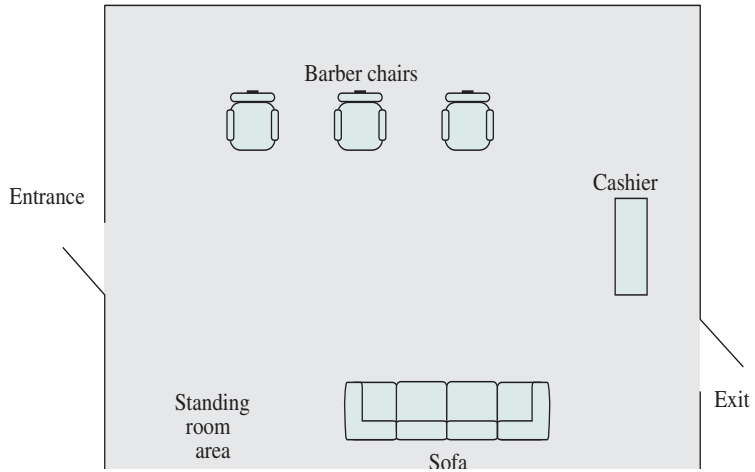
- **sofa (4) & max_capacity (20)**
- Three chairs (three barbers)—**barber_chair (3)**
  - Decremented (incremented) when client sitting (getting up)
  - Clients will not wake up from sofa if not signaled by barber
- Barber sleeping when no client sited—**cust_ready (0)**
- Clients remaining sitting till cut is over– **finished (0)**
- Barber to not invite other client before current one has left the chair — **leave_b_chair (0)**
- Client to pay (signal) the cashier (who waits for money)—**payment (0)**
- Cashier to give (signal) the receipt to client who waits for it after paying—**receipt(0)**
- Only 3 people performing work in the shop (coordination btw barber and cashier role)—**coord (3)**

Barbershop Solution

```
customer:

Wait(max_capacity);
Enter_shop();
Wait(sofa);
Sit();
Wait(barber_chair);
Get_up();
Signal(sofa);
Sit_barber_chair();
Signal(cust_ready);
Wait(finished);
Leave_barber_chair();
Signal(leave_b_chair);
Pay();
Signal(payment);
Wait(receipt);
Exit_shop();
Signal(max_capacity);
```

```
barber:

While (true) {
 Wait(cust_ready);
 Wait(coord);
 cut_hair();
 signal(coord);
 signal(finished);
 wait(leave_b_chair);
 signal(barber_chair);
}
```

```
cashier:

While (true) {
 wait(payment);
 wait(coord);
 accept_payment();
 signal(coord);
 signal(receipt);
}
```



Barbershop Solution: Unfair

```
customer:

Wait(max_capacity);
Enter_shop();
Wait(sofa);
Sit();
Wait(barber_chair);
Get_up();
Signal(sofa);
Sit_barber_chair();
Signal(cust_ready);
Wait(finished);
Leave_barber_chair();
Signal(leave_b_chair);
Pay();
Signal(payment);
Wait(receipt);
Exit_shop();
Signal(max_capacity);
```

```
barber:

While (true) {
 Wait(cust_ready);
 Wait(coord);
 cut_hair();
 signal(coord);
 signal(finished);
 wait(leave_b_chair);
 signal(barber_chair);
}
```

```
cashier:

While (true) {
 wait(payment);
 wait(coord);
 accept_payment();
 signal(coord);
 signal(receipt);
}
```

- Idea: use different queues per chair

Fair barbershop semaphores

- Add variable **count—**client number
- **Mutex1 (1) to protect access to count**
- sofa (4) & max_capacity (20)
- barber_chair (3)
- Barber sleeping when no client sited—cust_ready (0)
- Customers remaining sitting till cut is over**– Array of sem. finished [50] (0)**
  - **Wait(finished[custnr]) by customer**
  - **Signal(finished[b_cust]) by barber to release the correct customer**
- Queue1 protected by **mutex2 (1)**
  - Customers put their number on the queue **Enqueue1(custnr)** before signaling with cust_ready
  - Barber **Dequeue(b_cust)** to get the topmost customer and set b_cust=cust_nr
- leave_b_chair (0)
- payment (0)
- receipt(0)
- coord (3)

Fair Barbershop Solution

```
customer:

Int custnr;

Wait(max_capacity);
Wait(mutex1);
Count++;
custnr = Count;
Signal(mutex1);
Wait(sofa);
Wait(barber_chair);
Signal(sofa);
Wait(mutex2);
enqueue1(custnr)
Signal(cust_ready);
Signal(mutex2);
Wait(finished[custnr]);
Signal(leave_b_chair);
Signal(payment);
Wait(receipt);
Signal(max_capacity);
```

```
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);
}
```

```
cashier:

While (true) {
 wait(payment);
 wait(coord);
 accept_payment();
 signal(coord);
 signal(receipt);
}
```

Message Passing Systems

- Enforce mutual exclusion
- Exchange information

- Typical functionalities of MP systems:

  `send (destination, message)`

  `receive (source, message)`

Synchronization

- Communication of a message implies synchronization
  - msg cannot be received unless has been sent
- Send and receive primitives may be blocking
  - send: process waiting for message to be received
  - receive: blocked until msg arrives or continues abandoning the reception
- Blocking send, blocking receive (redezvous)
- Nonblocking send, blocking receive (sender continues on)
- Nonblocking send, nonblocking receive (abandon or test for arrival)

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 expected from
  - 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

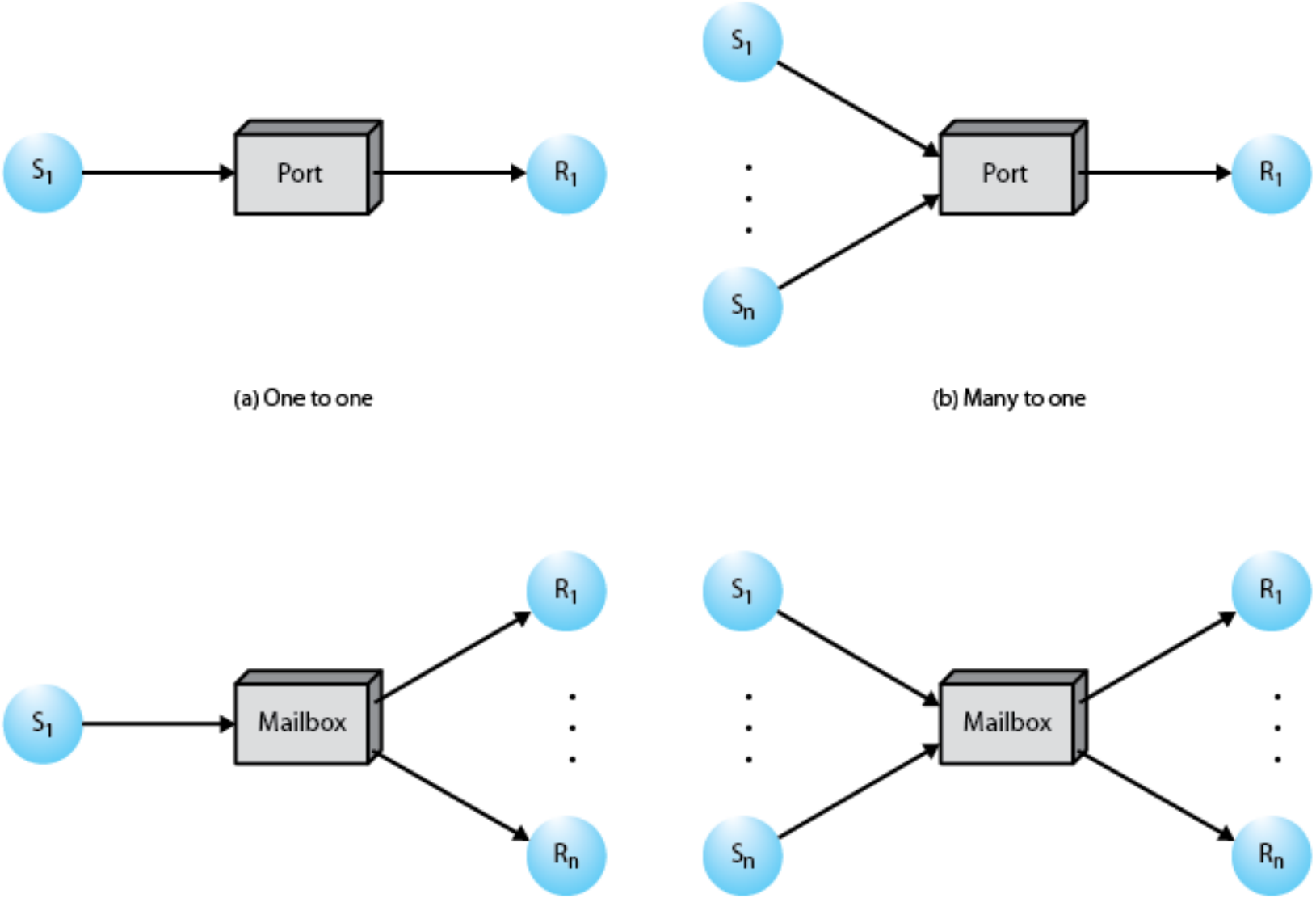


Figure 5.18 Indirect Process Communication

Message Format

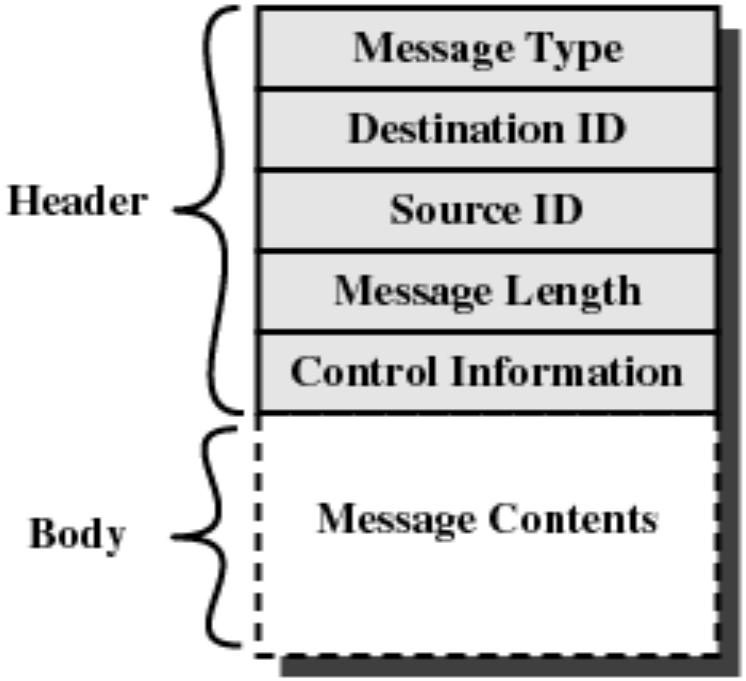


Figure 5.19 General Message Format

Mutual Exclusion with Mailboxes

- Unblocking send, blocking receive
- One mailbox initialized with 1 empty message in it

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

Producer consumer with mailboxes

- Two mailboxes:
- Mayconsume
  - contains messages sent by producer after generating data
  - Serves as buffer

- Mayproduce
  - contains message generated by consumer each time it consumes

```
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 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Readers/Writers Problem

- Problem:
  - Data shared among processes (e.g. file)
  - 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/Writers Problem

- Case 1: Readers have priority—retain the shared file
  - Semaphore wsem for mutual exclusion
  - If other readers enabled, new readers should pass -> variable readcount and semaphore x

```
/* 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 5.22 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

Readers/Writers Problem

- Case 1: Readers retain the data as much as they need to => risk of starvation for writers
- Case 2: Writers have priority
  - Semaphore rsem blocks readers if there is a writer wanting to write
  - Variable writecount controlling setting of rsem
  - Semaphore y controlling updating of writecount
  - Semaphore z for readers queue

Readers/Writers Problem

Readers only in the system	• *wsem* set • no queues
Writers only in the system	• *wsem* and *rsem* set • writers queue on *wsem*
Both readers and writers with read first	• *wsem* set by reader • *rsem* set by writer • all writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*
Both readers and writers with write first	• *wsem* set by writer • *rsem* set by writer • writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*

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