Operating systems

Lecture 4, 5, 6

Michal Vrábel, 06/11/2019 - 20/11/2019

Monitors

- synchronization by the use of **condition variables** that are
  - contained within the monitor and
  - accessible only within the monitor.
- **Condition variables**
  - special data type in monitors,
  - operated on by two functions:
  - **cwait(c)**: Suspend execution of the calling process on condition c.
    - The monitor is now available for use by another process.
  - **csignal(c)**: Resume execution of some process blocked after a cwait on the same condition.
    - If there are several such processes, choose one of them;
    - If there is no such process, do nothing.
- **monitor wait and signal operations are different from those for the semaphore**
  - If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.
- Reading: https://en.wikipedia.org/wiki/Monitor_(synchronization)
- Practical example: https://www.baeldung.com/java-wait-notify

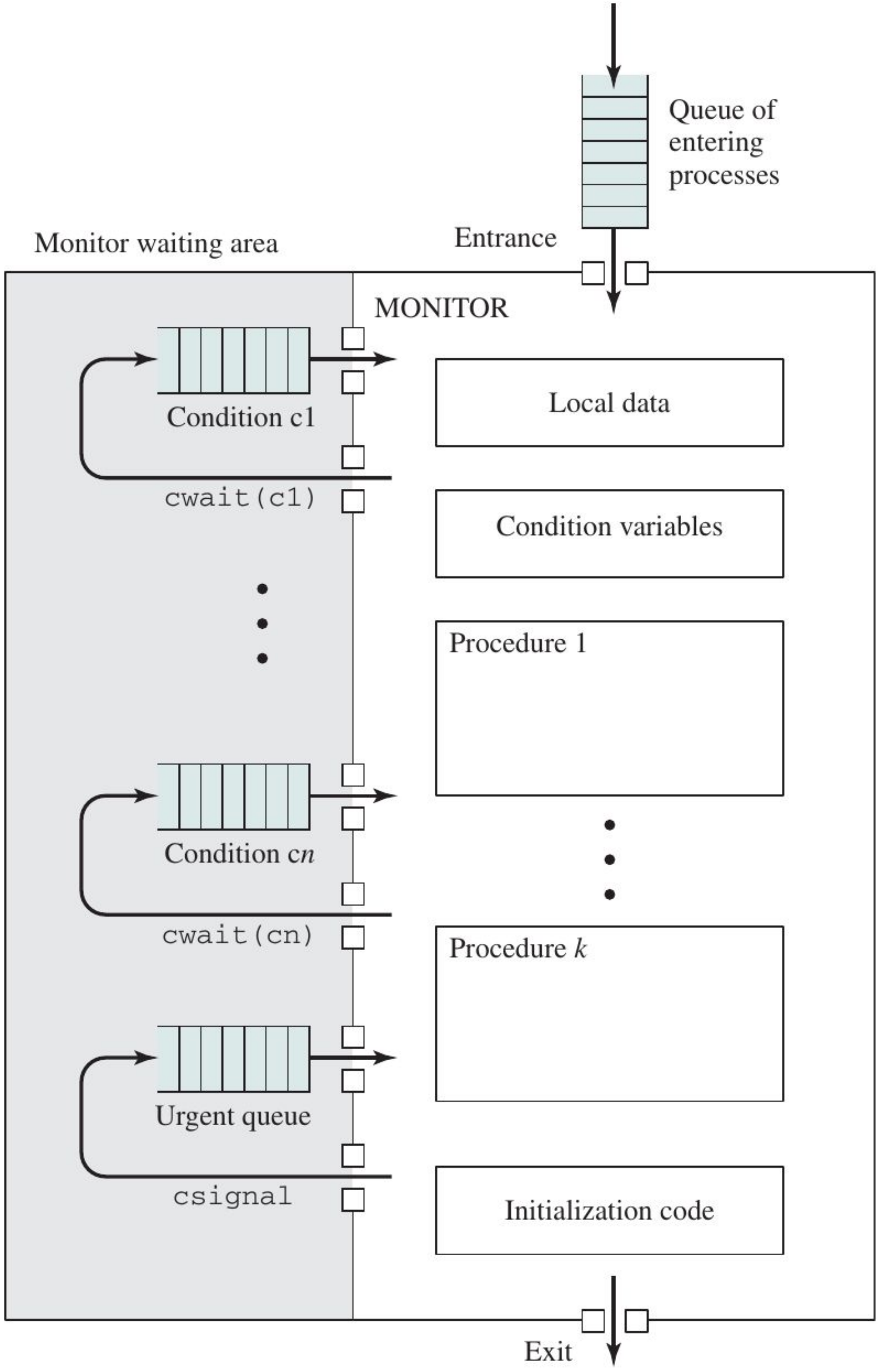


Figure 5.15 **Structure of a Monitor**

https://en.wikipedia.org/wiki/Monitor_(synchronization)

Monitors (from wikipedia)

- In concurrent programming
- A synchronization construct
- allows threads to have both **mutual exclusion** and the **ability to wait (block) for a certain condition** to become false.
- have a mechanism for signaling other threads that their condition has been met
- consists of a **mutex (lock) object** and **condition variables**.
    - A condition variable - a container of threads that are waiting for a certain condition.
- provide a **mechanism for threads to temporarily give up exclusive access** in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

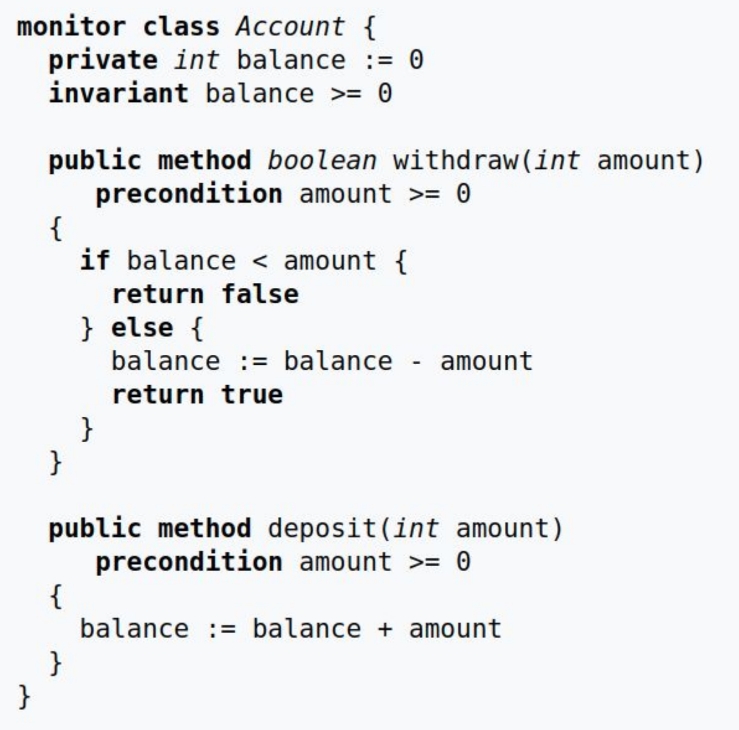
https://en.wikipedia.org/wiki/Monitor_(synchronization)

Monitors (from wikipedia, another definition)

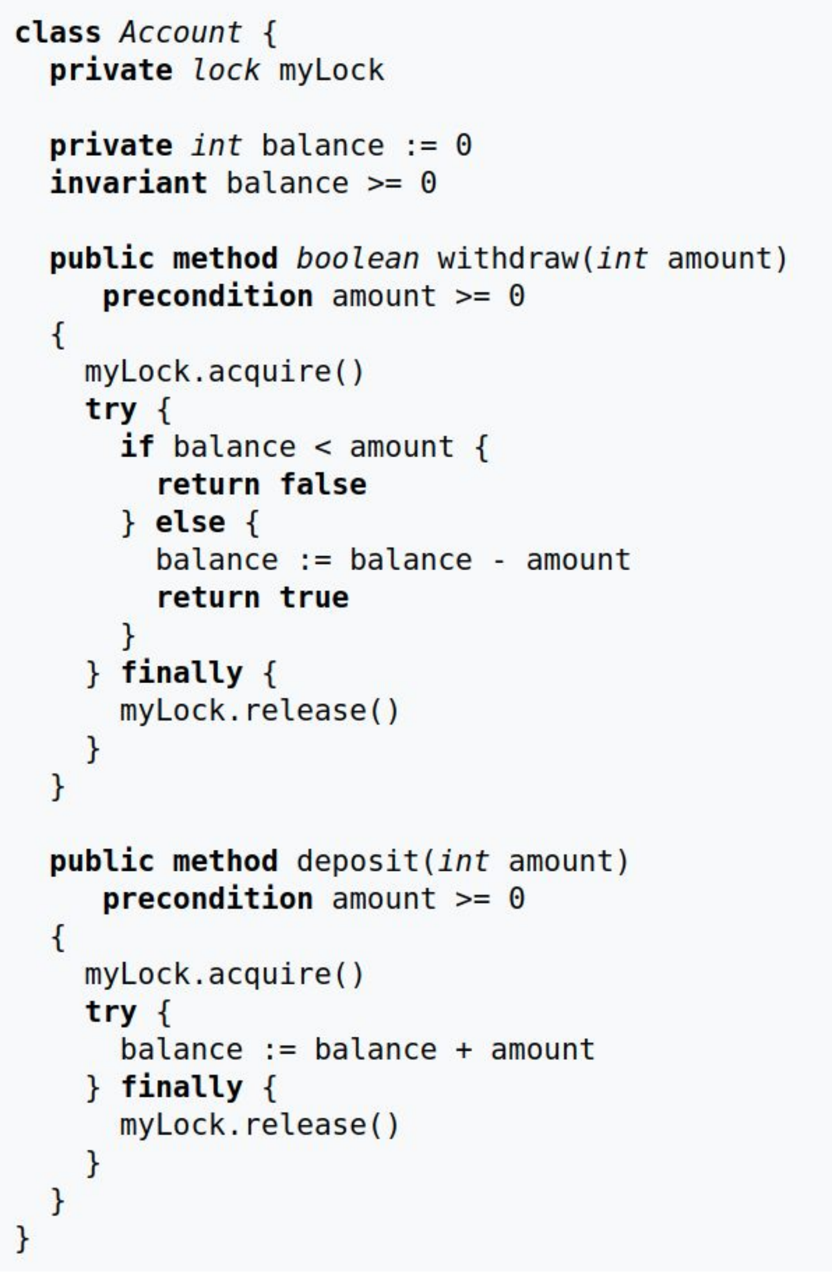
- a thread-safe class, object, or module that wraps around a mutex in order to safely allow access to a method or variable by more than one thread.
- its methods are executed with mutual exclusion:
  - **At each point in time, at most one thread may be executing any of its methods.**
  - By using one or more condition variables it can also provide the **ability for threads to wait on a certain condition** (thus using the above definition of a "monitor").
  - "thread-safe object/class/module".

https://en.wikipedia.org/wiki/Monitor_(synchronization)

Monitors (from wikipedia, another definition)

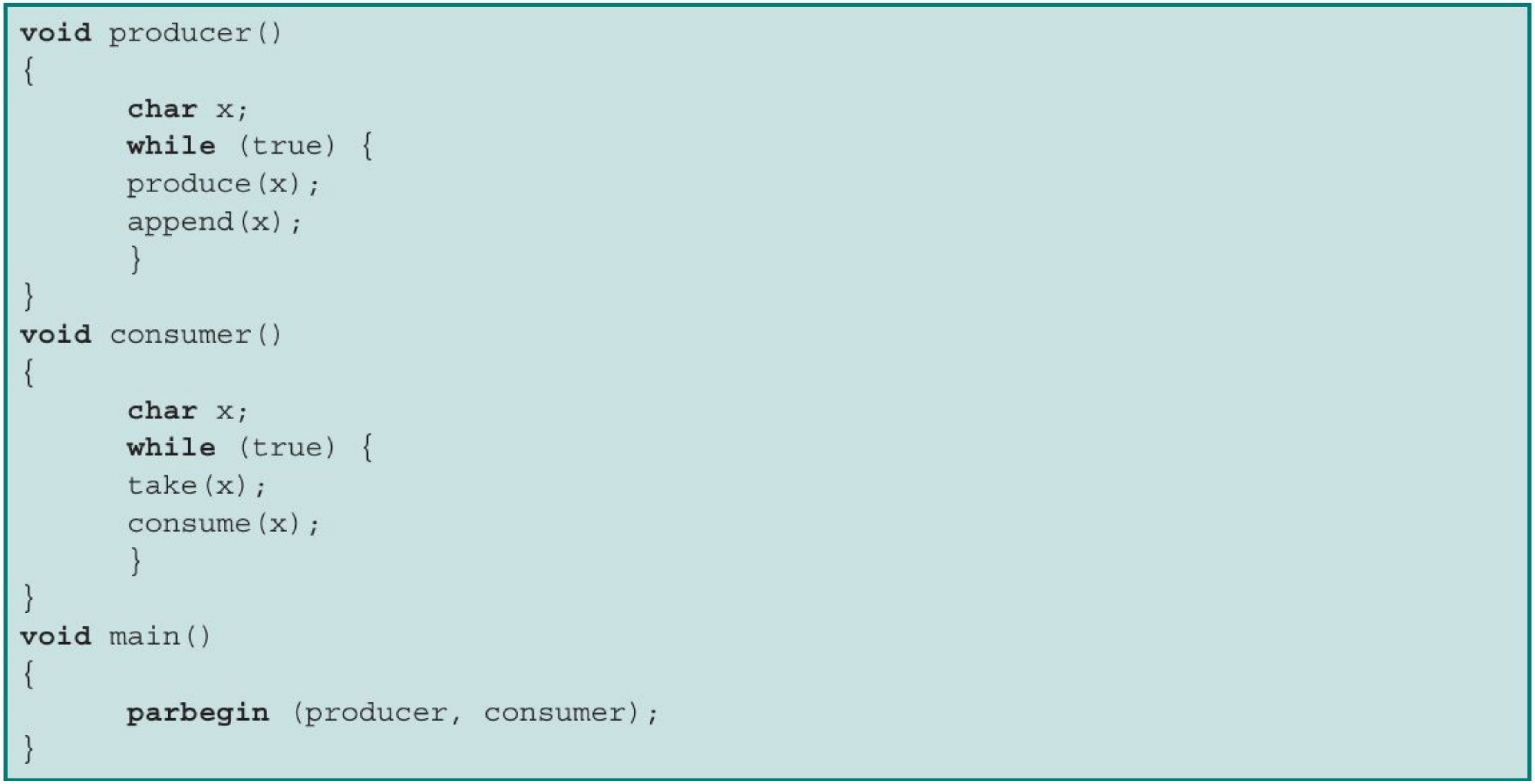


```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    if balance < amount {
      return false
    } else {
      balance := balance - amount
      return true
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
  }
}
```



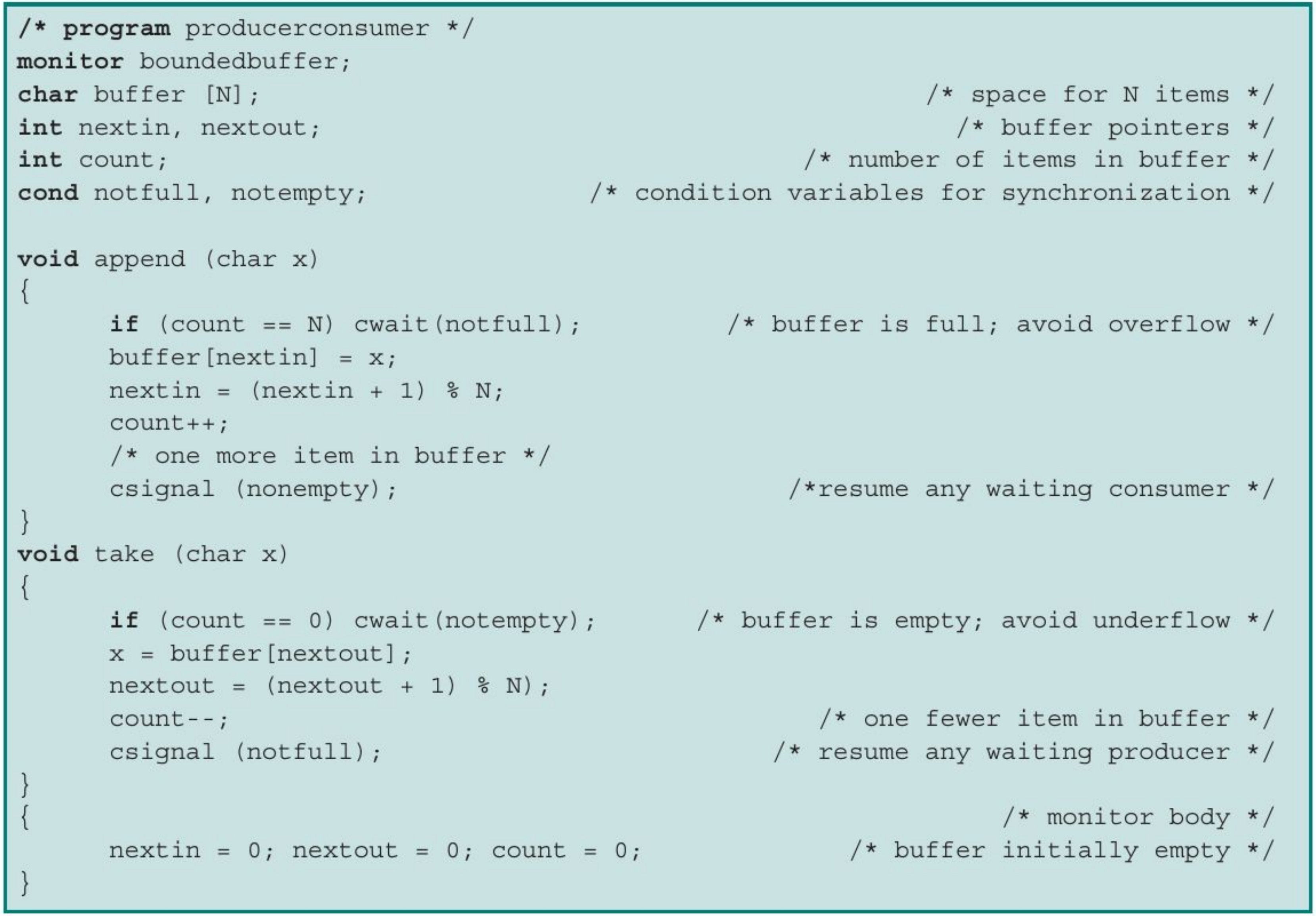
```
class Account {
  private lock myLock

  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      if balance < amount {
        return false
      } else {
        balance := balance - amount
        return true
      }
    } finally {
      myLock.release()
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      balance := balance + amount
    } finally {
      myLock.release()
    }
  }
}
```

Monitors – usage of a 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);
}
```

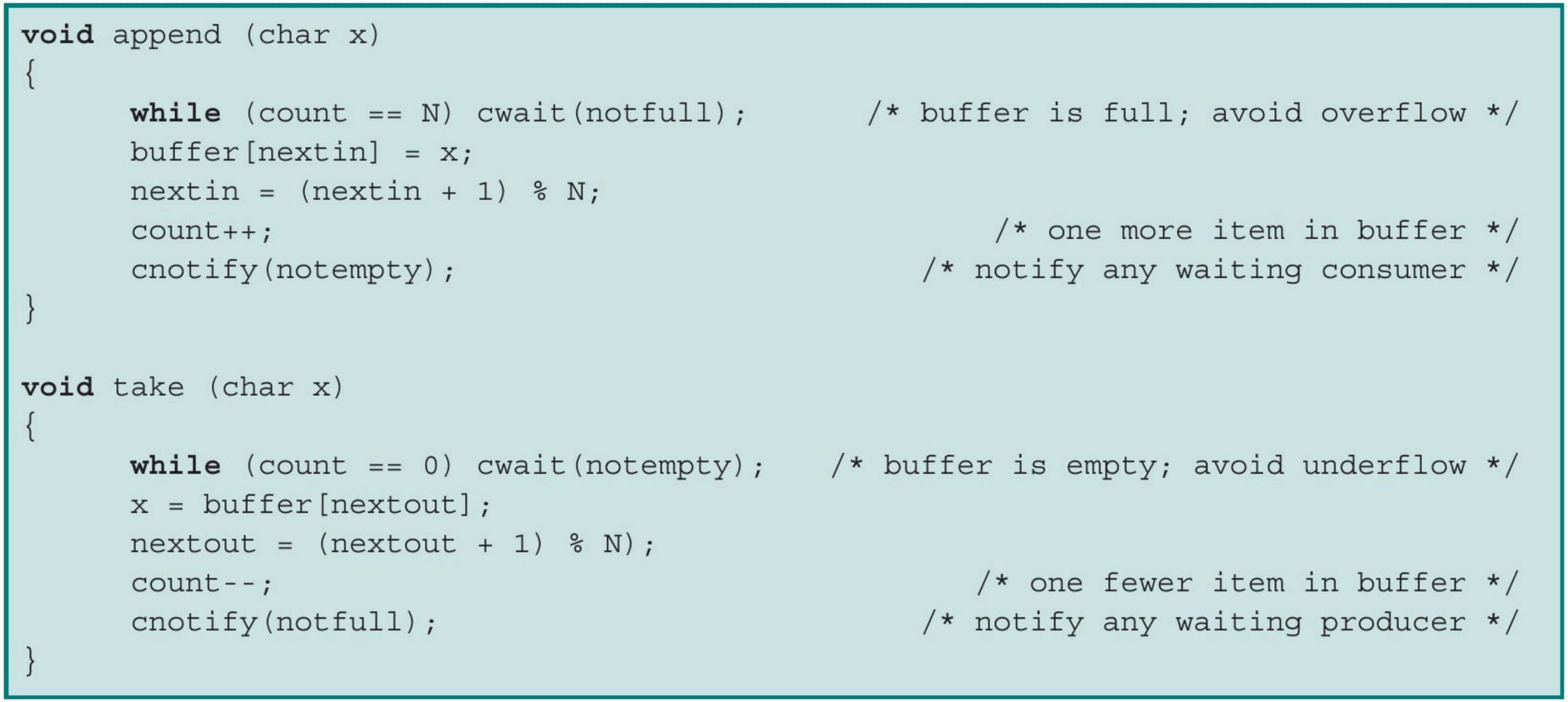
Monitors – usage of a monitor



```
/* 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 (nonempty);                          /*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 */
}
```

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

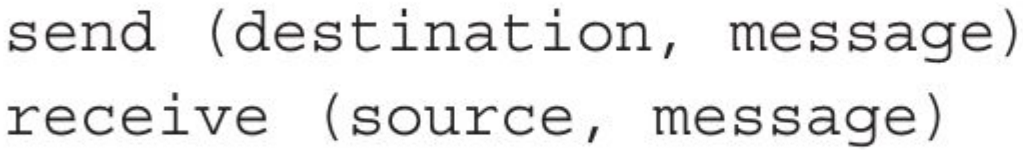
Monitors – alternate model



```
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 5.17 Bounded-Buffer Monitor Code for Mesa Monitor

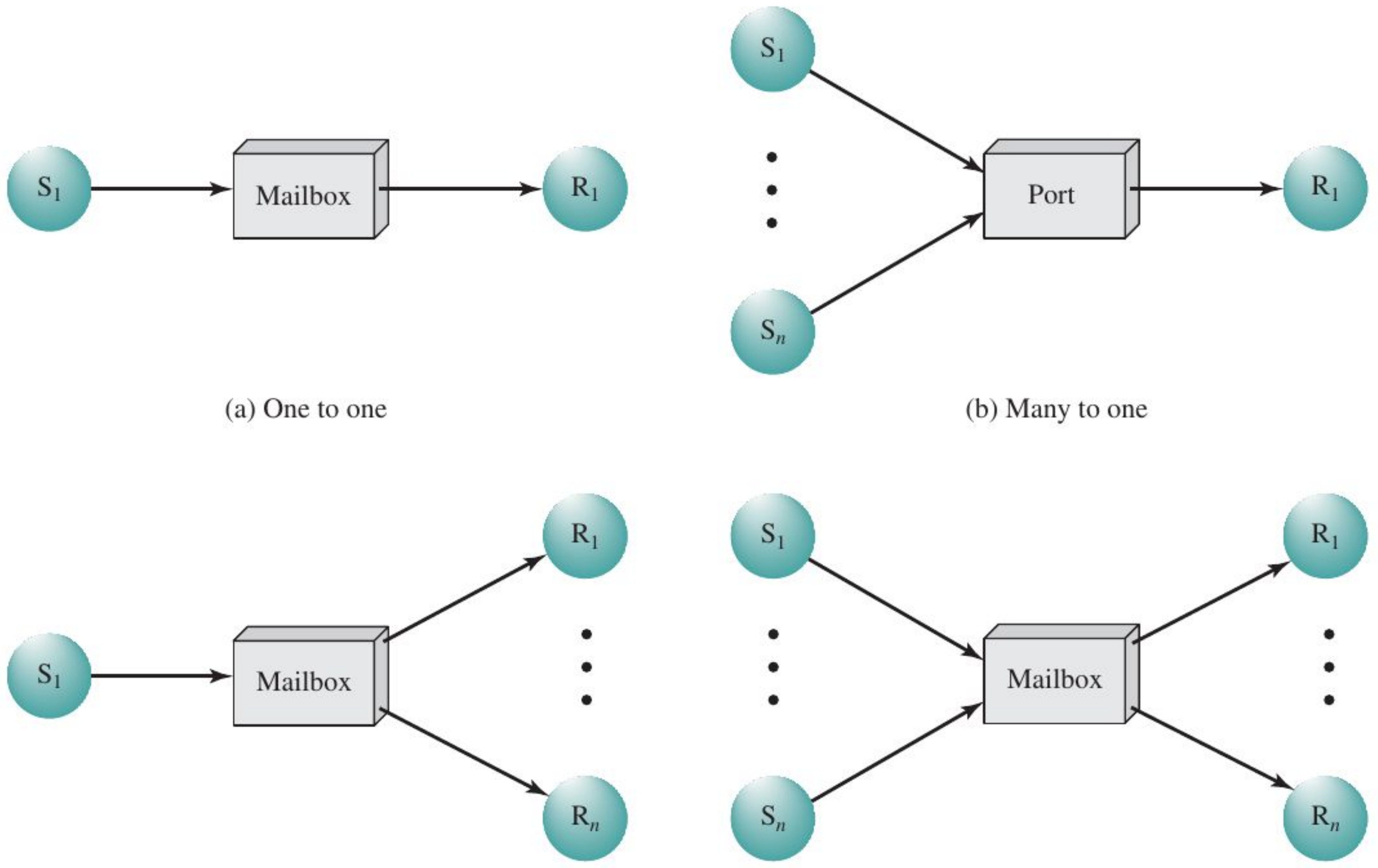
Message passing



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

- **Blocking send, blocking receive:**
  - Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a rendezvous.
  - This combination allows for tight synchronization between processes.
- **Nonblocking send, blocking receive:**
  - Although the sender may continue on, the receiver is blocked until the requested message arrives.
  - This is probably the most useful combination.
  - It allows a process to send one or more messages to a variety of destinations as quickly as possible.
  - A process that must receive a message before it can do useful work needs to be blocked until such a message arrives.
  - An example is **a server process** that exists to provide a service or resource to other processes.
- **Nonblocking send, nonblocking receive:**
  - Neither party is required to wait.

Message passing - addresing



(a) One to one

(b) Many to one

(c) One to many

(d) Many to many

Figure 5.18 **Indirect Process Communication**

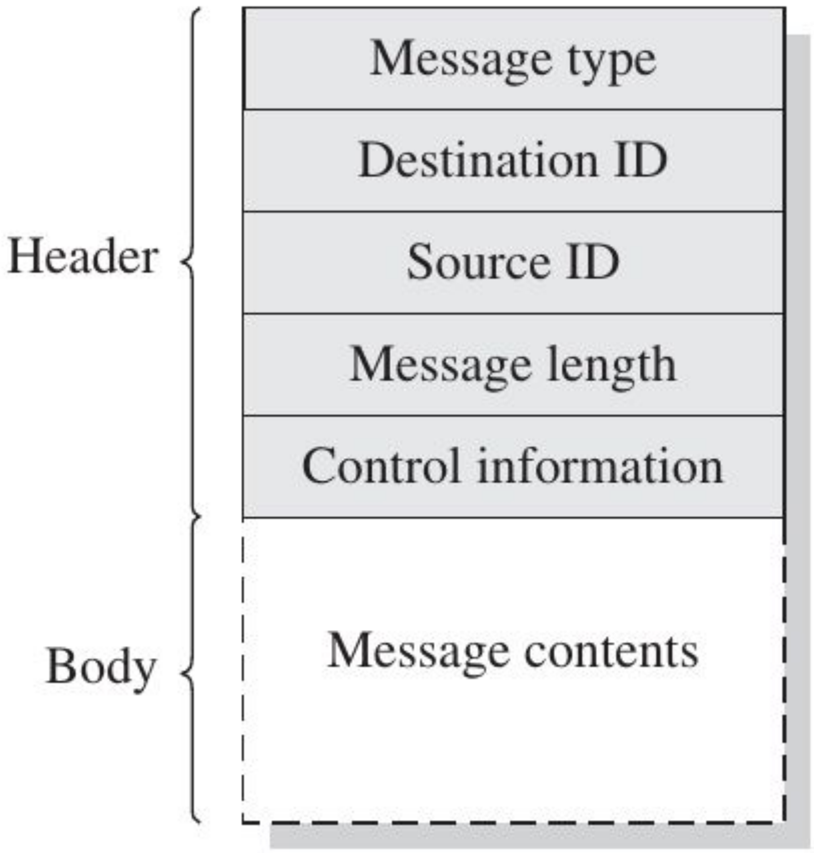
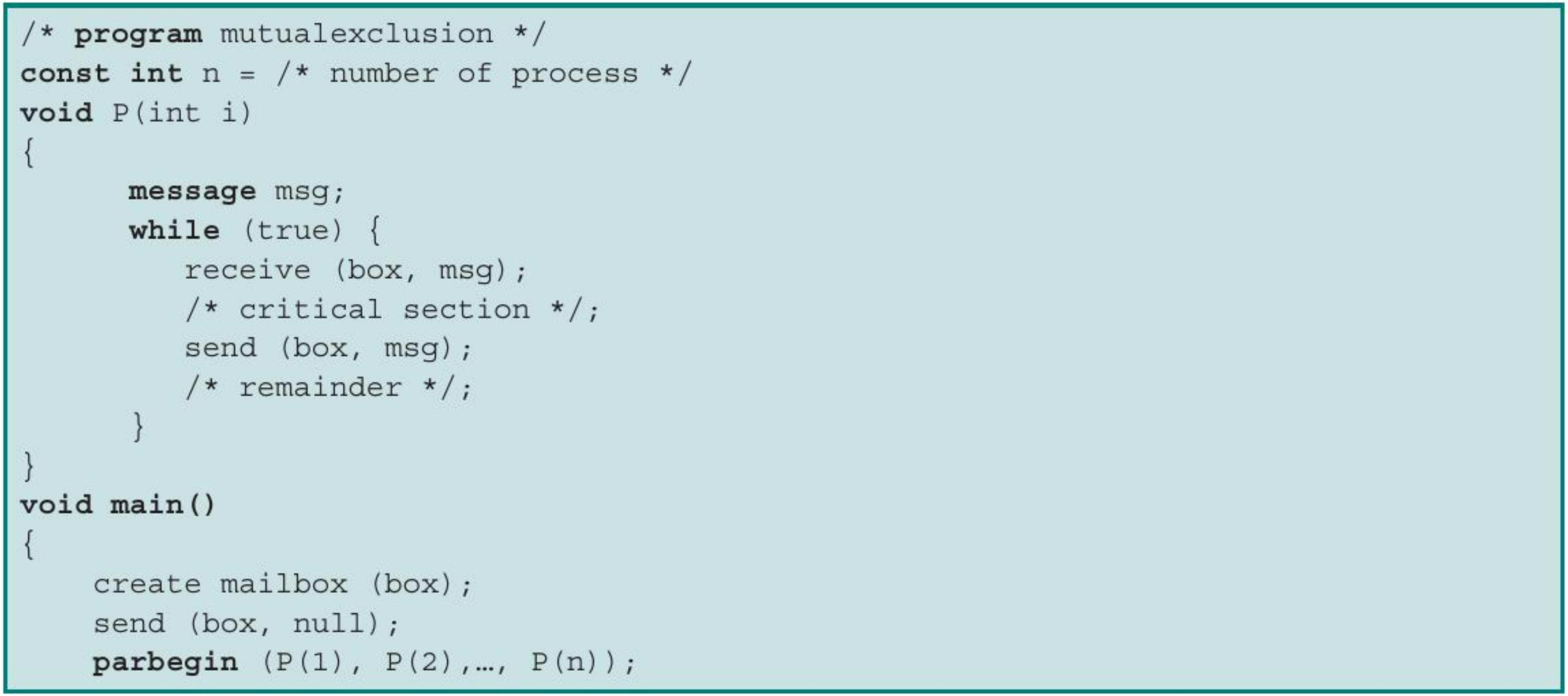


Figure 5.19 **General Message Format**



```
/* program mutualexclusion */
const int n = /* number of process */
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

```
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

- Common design problem
- Similar to producer / consumer problem
- There is a **data area shared among a number of processes**.
- The data area could be a file, a block of main memory, or even a bank of processor registers.
- There are a number of processes that only
  - read the data area (readers)
  - write to the data area (writers).
- Conditions
  - 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

- **Readers**
  - processes that are **not required to exclude one another**
  - Do not also write to the data area
- **Writers**
  - processes that are **required to exclude all other processes**, readers and writers alike.
  - Do not read the data area while writing
- **Producer / consumer problem is not readers / writers**
  - **producer is not just a writer**
    - must read queue pointers to determine where to write the next item,
    - must determine if the buffer is full
  - **consumer is not just a reader**
    - must adjust the queue pointers to show that it has removed a unit from the buffer

Readers / Writers problem

- **Readers Have Priority**
  - The writer process is simple
  - As long as one writer is accessing the shared data area, no other writers and no readers may access it
  - allows multiple readers
  - when there are no readers reading, the first reader that attempts to read should wait
  - When there is already at least one reader reading, subsequent readers need not wait before entering
  - writers are subject to starvation - Once a single reader has begun to access the data area, it is possible for readers to retain
- **Writers 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;~~ readcount--;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

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

- **wsem** to enforces mutual exclusion
- when there are no readers reading, the first reader that attempts to read should wait on **wsem**
- The **global variable readcount** is used to keep track of the number of readers,
- the **semaphore x** is used to assure that readcount is updated properly

Figure 5.22 **A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority**

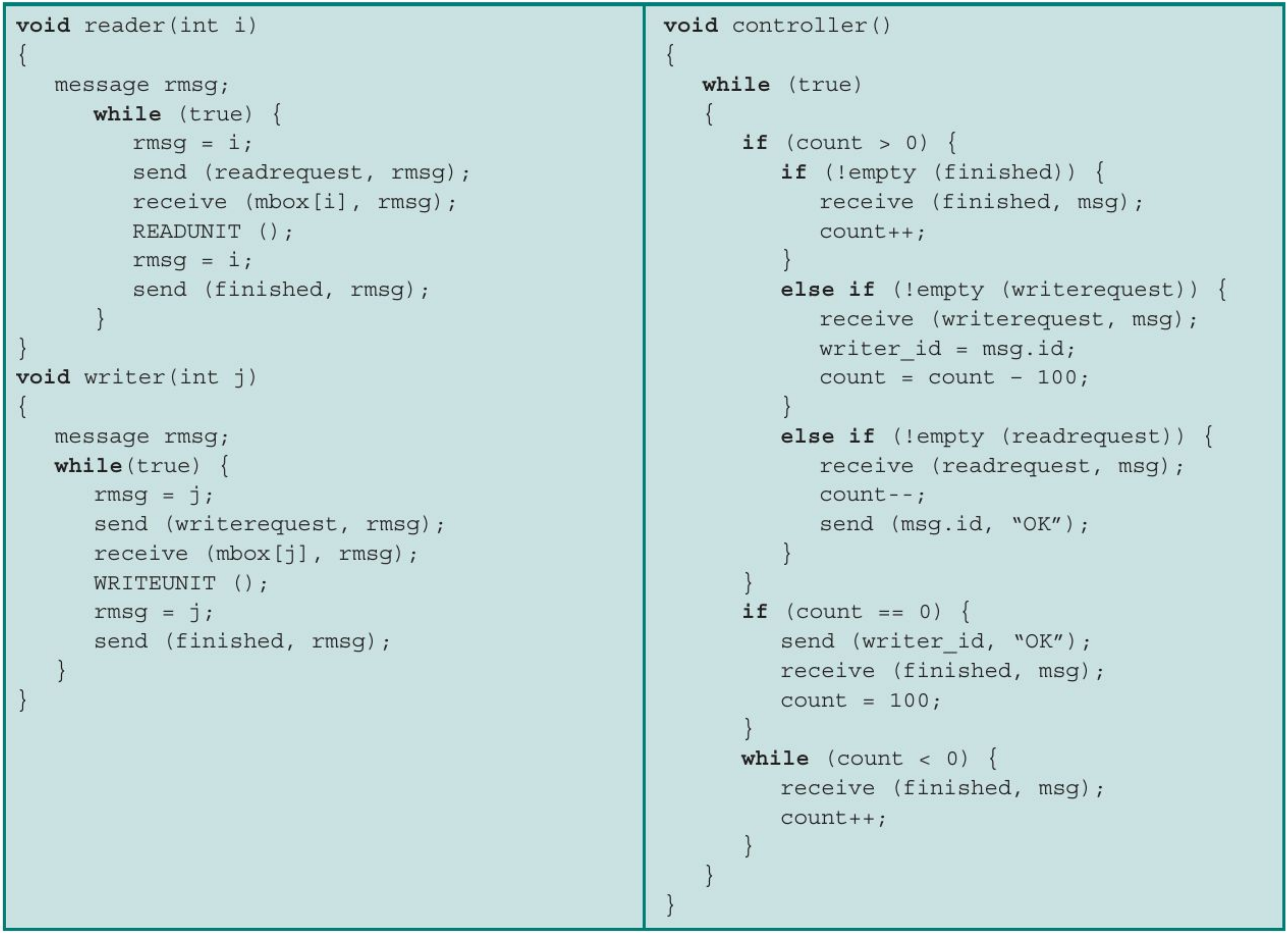
Readers / Writers problem

- **Readers Have Priority**
  - …
- **Writers Have Priority**
  - no new readers are allowed access to the data area once at least one writer has declared a desire to write
  - More complicated
  - Solvable via message passing
  - See the book:

5.6 / READERS/WRITERS PROBLEM 241

```
/* program readersandwriters */
int readcount,writecount;
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 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

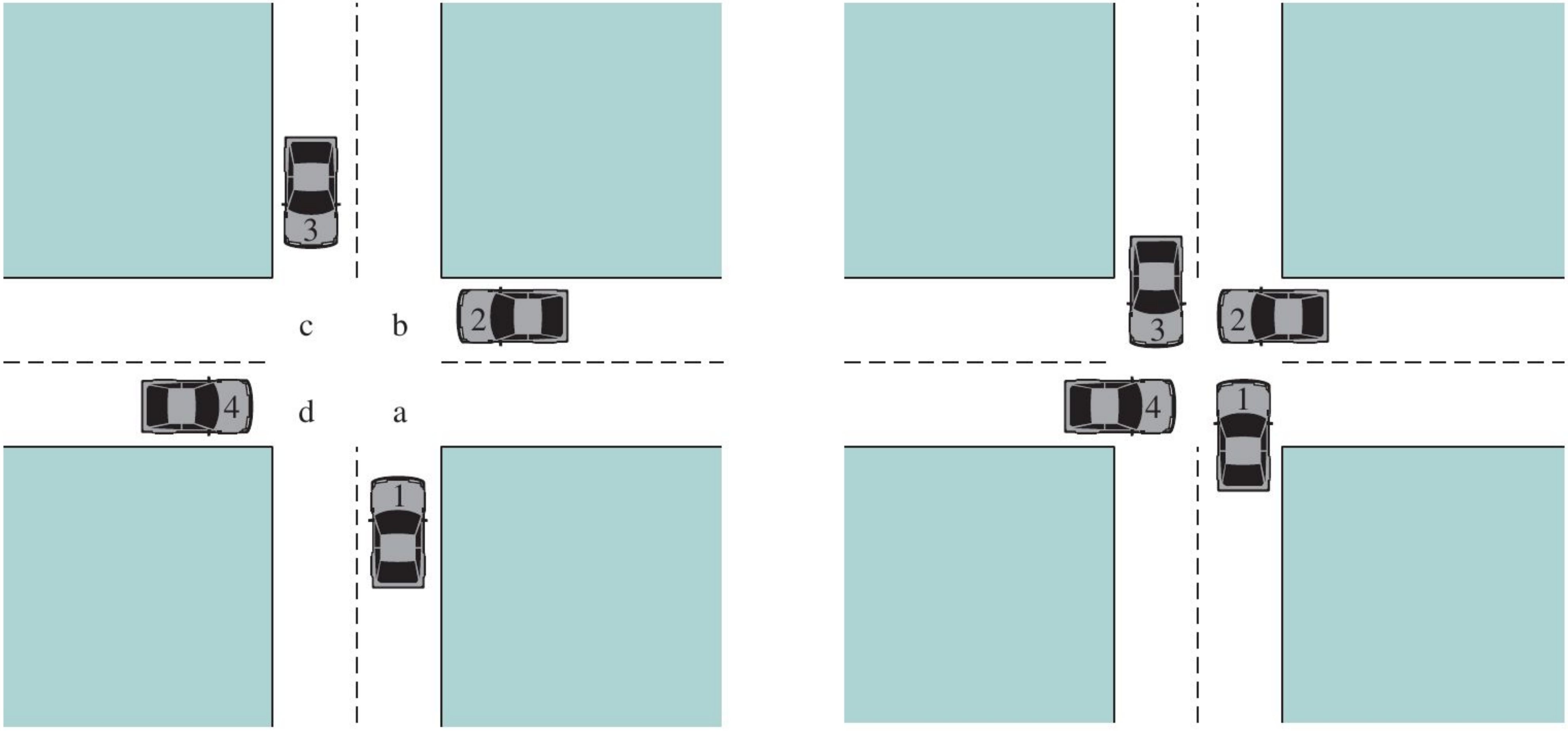


```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
}
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = 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 5.24 A Solution to the Readers/Writers Problem Using Message Passing

Deadlock



(a) Deadlock possible (b) Deadlock

Figure 6.1 Illustration of Deadlock

Conditions for deadlock

- **Mutual exclusion** - Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
- **Hold and wait** - A process may hold allocated resources while awaiting assignment of other resources.
- **No preemption** - No resource can be forcibly removed from a process holding it.
- **Circular wait** - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention (prevent at least one condition)	*Hold & wait* Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
	Circular wait	Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance Resource Allocation Denial	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

Table 6.4 Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise