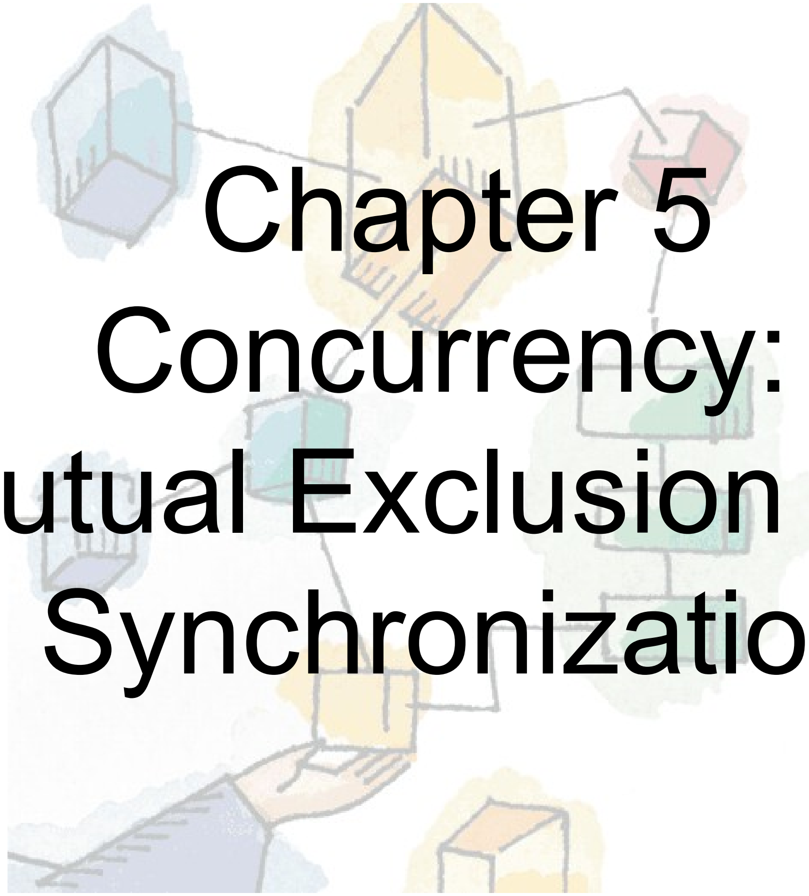
Operating Systems:
Internals and Design Principles, 8/E
William Stallings

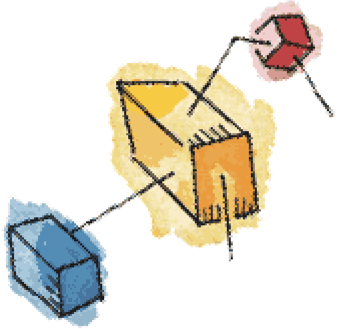


Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Patricia Roy
Manatee Community College, Venice, FL

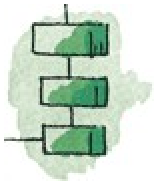
The Outline

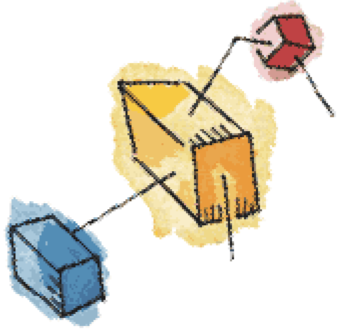
- Principle of Concurrency
  - Race Condition, OS Concerns, requirement for Mutual exclusion
- Mutual exclusion: Hardware support
  - Interrupt disabling, special machine instructions
- Semaphores
- Monitors
- Message Passing
- Readers / Writers Problem



Overview

- In the previous chapters, we were concerning about the management of threads and processes.
  - Multiprogramming (multiple processes, single processor)
  - Multiprocessing (multiple processes, multiple processors)
  - Distributed processing (multiple processes, multiple processors, distributed system)
- The fundamental behind all of these is:
  - Concurrency

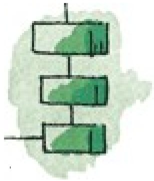


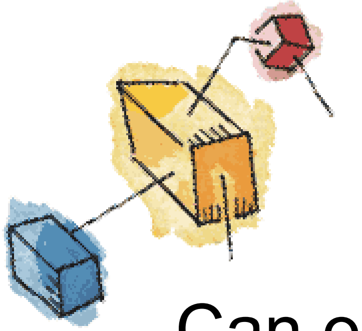


What is Concurrency?

- Property of handling many different tasks at the same time (concurrent) by the user or the system itself
- Handles:
  - Communication among processes
  - Sharing and competing for resources (I/O, files, memory)
  - Synchronization of activities of multiple processes
  - Allocation of processor time to processes

- Arises in multiprocessing, distributed processing, & single-processor

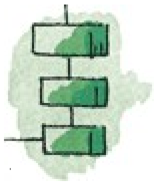




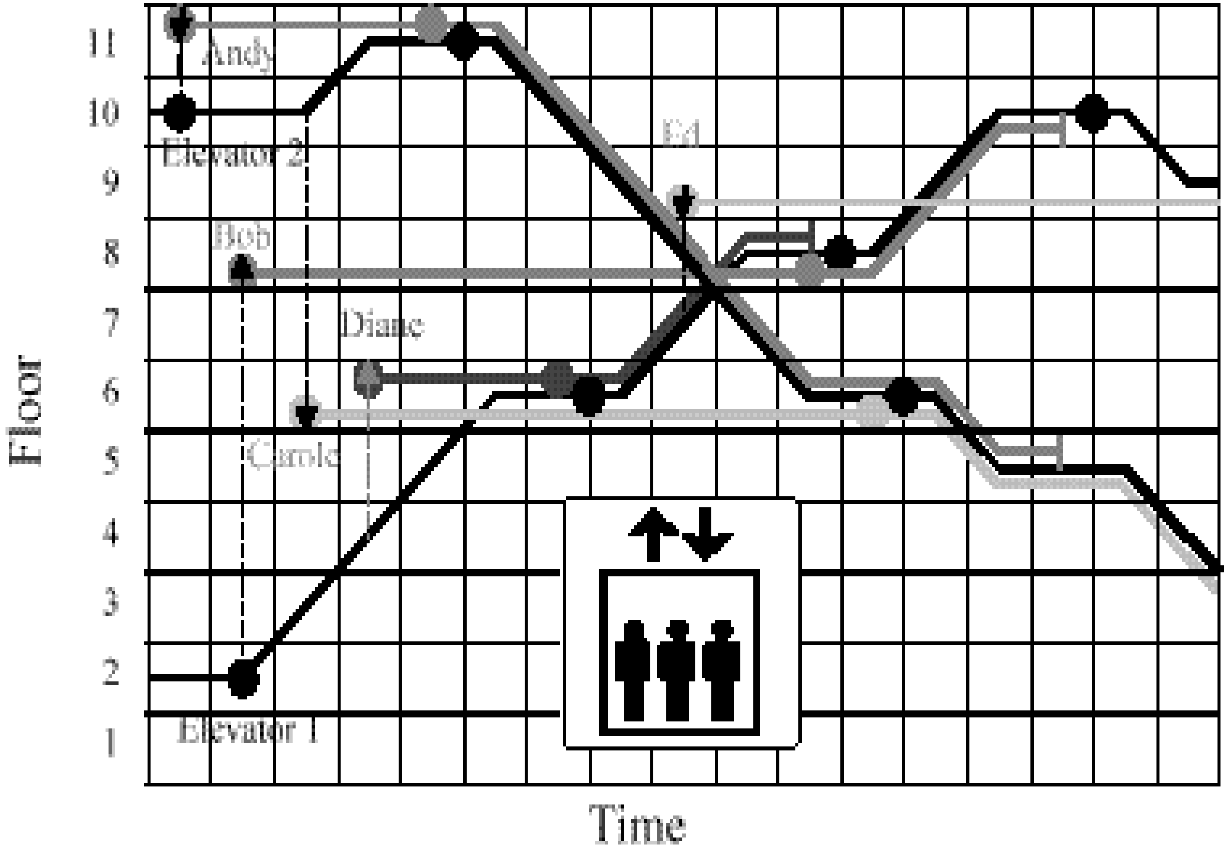
Concurrency

Can occur in three different contexts:

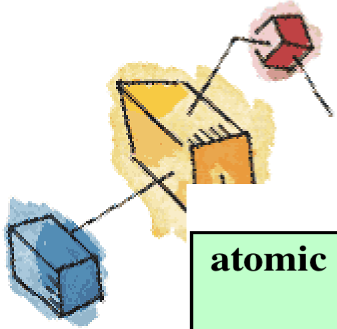
- Multiple applications
  - Multiprogramming
  - to allow processing time to be shared among a number of applications
- Structured applications
  - Extension of modular design, a set of concurrent processes
- Operating system structure
  - The OS itself often implemented as a set of processes or threads



Example: Elevator System

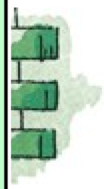


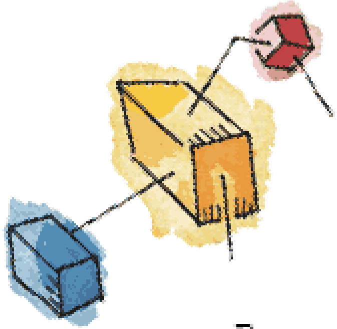
Designed to control a group of elevators at one location in a building



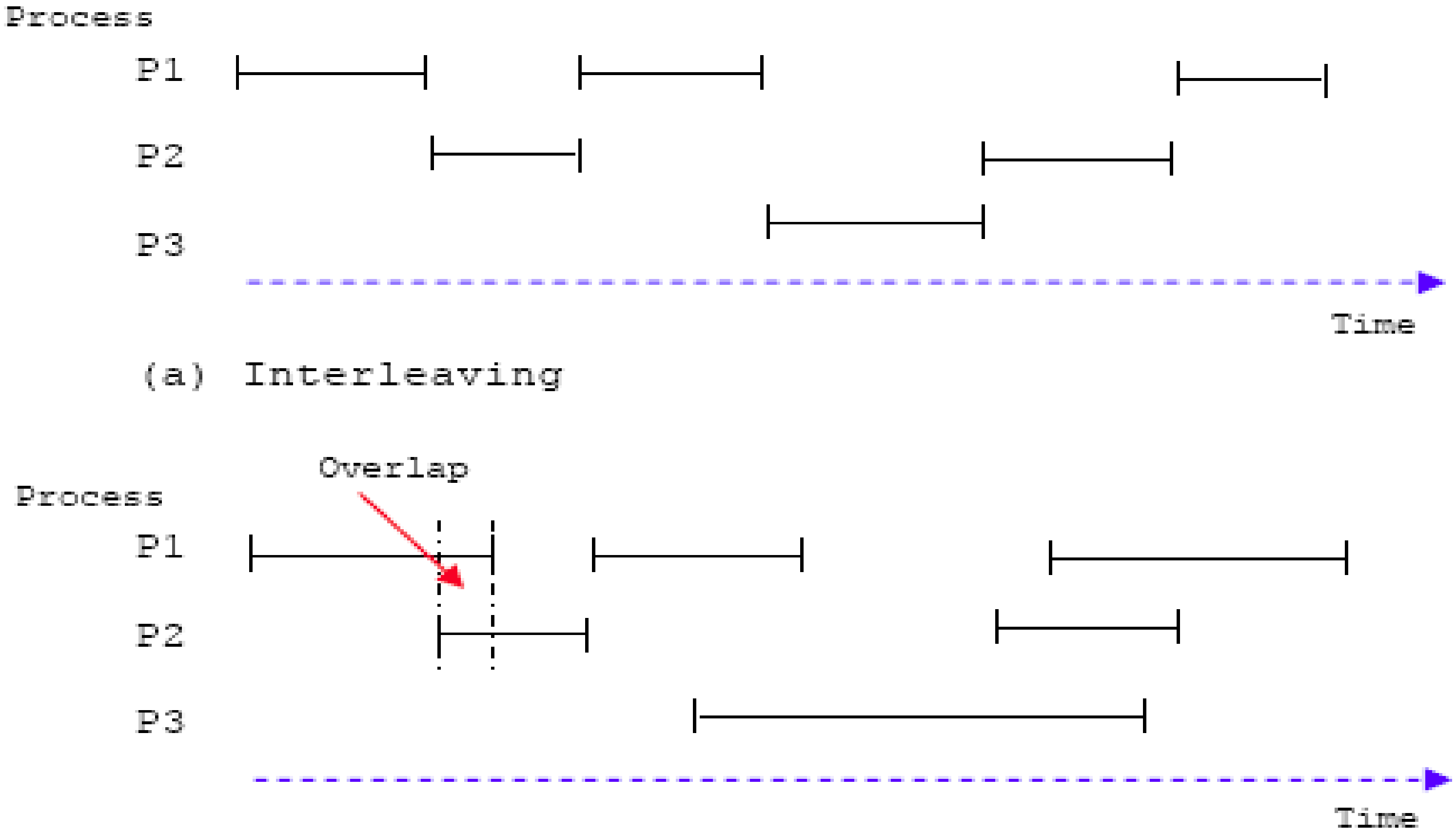
Key Terms

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
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.



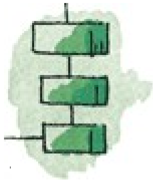


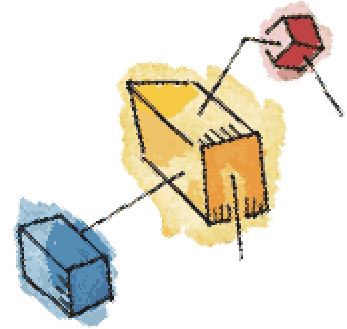
Principle of Concurrency



(a) Interleaving

(b) Interleaving and overlapping

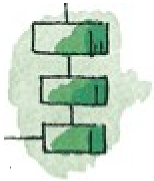


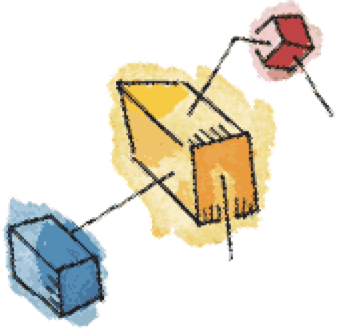


Principle of Concurrency

- Interleaving and overlapping
  - Interleave = For both uniprocessor & multiprocessor
  - Overlap = ONLY for multiprocessor
  - Can be viewed as examples of concurrent process
  - Both present the same problem

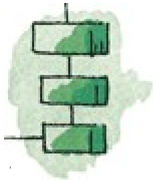
- The uniprocessor – the relative speed of execution of processes cannot be predicted because
  - It depends on activities of other processes
  - The ways the OS handles the interrupt
  - Scheduling policies of the OS

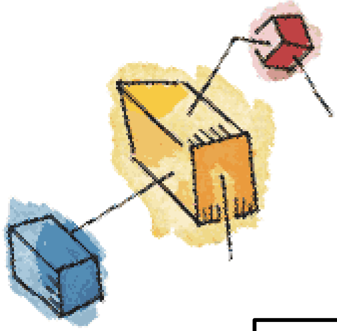




Difficulties of Concurrency

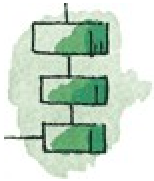
- Sharing of global resources is fraught with peril
  - e.g. two processes both make use of the same global variable, both performs read / write on the variable
  - the order of the result might be incorrect
- Difficult for the OS to manage the allocation of resources optimally
  - May lead to deadlock
- Difficult to locate programming errors
  - Because errors are not deterministic & reproducible



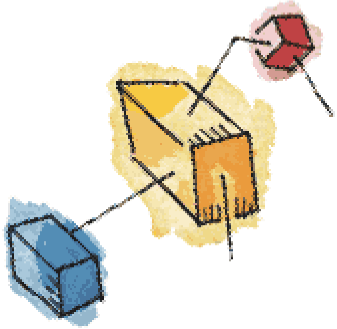


A Simple Example

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

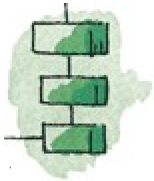


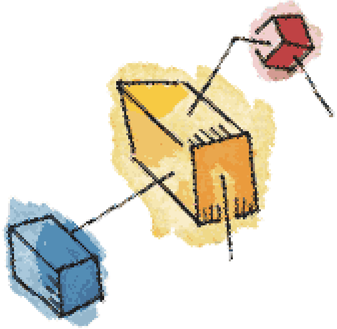
Can be called by any program repeatedly



Example

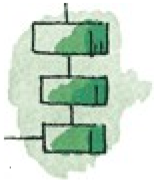
- Consider a single-processor multiprogramming system supporting a single user
- The user can jump from one application to another
- Each application uses the same keyboard for input & the same screen for output
- Share the same *echo* procedure – loaded into main memory and global to all applications
- Thus, only a single copy of the *echo* procedure is used to save space.



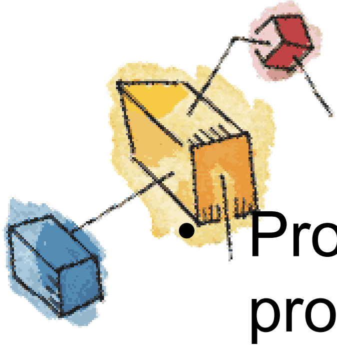


Possible Problem

- Process P1 invokes the *echo* procedure
- P1 is interrupted immediately after *getchar* returns and stores it in *chin*
- At this point, the most recent value stored is *x*
- P2 is activated, and invoke the echo procedure, execute and display a character *y*
- P1 is resumed. By this time, value x has been overwritten and therefore lost. Instead, *chin* contains value y which is transferred to chout and displayed
- x is lost, y is displayed twice!

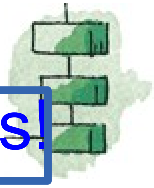


Solution

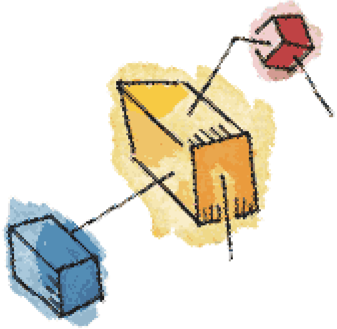


- Process P1 & P2 are both executing on a separate processor.
- P1 invokes the echo procedure
- While P1 is still inside the echo, P2 invokes the echo
- Because P1 is still inside the echo, P2 is blocked from entering the echo
- Therefore, P2 is suspended waiting for the echo to be available
- P1 exits the execution of echo, exits the procedure and proceed with other process.
- Immediately, P2 is resumed and begins executing the echo

X is not lost, both X & Y can be displayed!



KEY: Control the access to the shared resources!

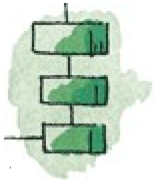


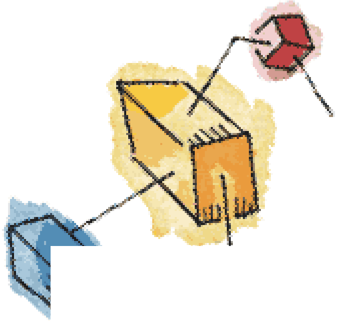
Race Condition

- Occurs when multiple processes or threads access resources without making sure the current one has already finished or not.

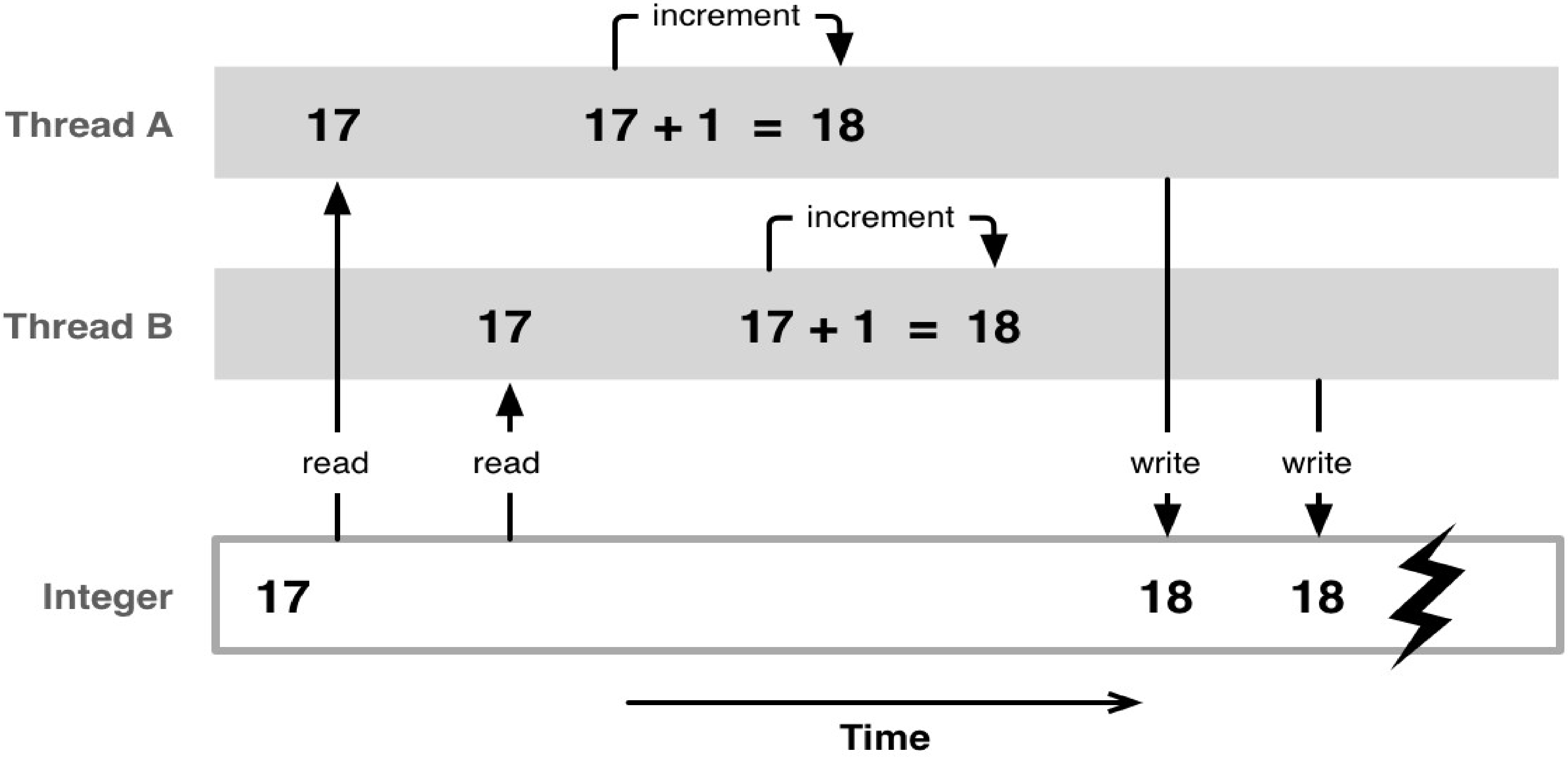
- The second thread might try to read/write when the first one is in the midst of it.

- The final result depends on the order of execution
    - the “loser” of the race is the process that updates last and will determine the final value of the variable





Race Condition



increment

Thread A

17

17 + 1 = 18

increment

Thread B

17

17 + 1 = 18

read

read

write

write

Integer

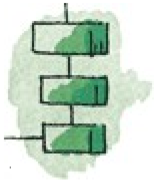
17

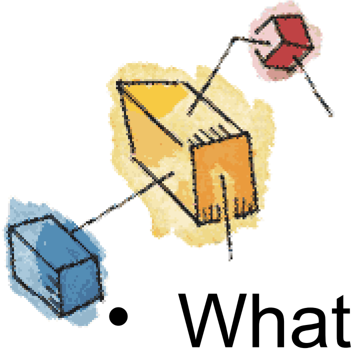
18

18

Time

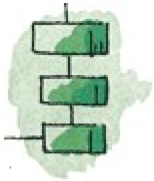
In order to prevent this, multiple threads need to access shared resources in a mutually exclusive way.

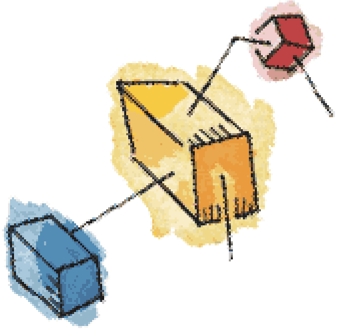




OS Concerns

- What design & management issues are raised by the existence of concurrency?
  - OS must keep track of various processes (using PCB)
  - OS must allocate and deallocate various resources for each active process
    - Processor time
    - Memory
    - Files
    - I/O devices

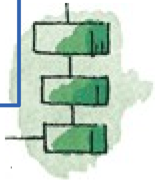




OS Concerns

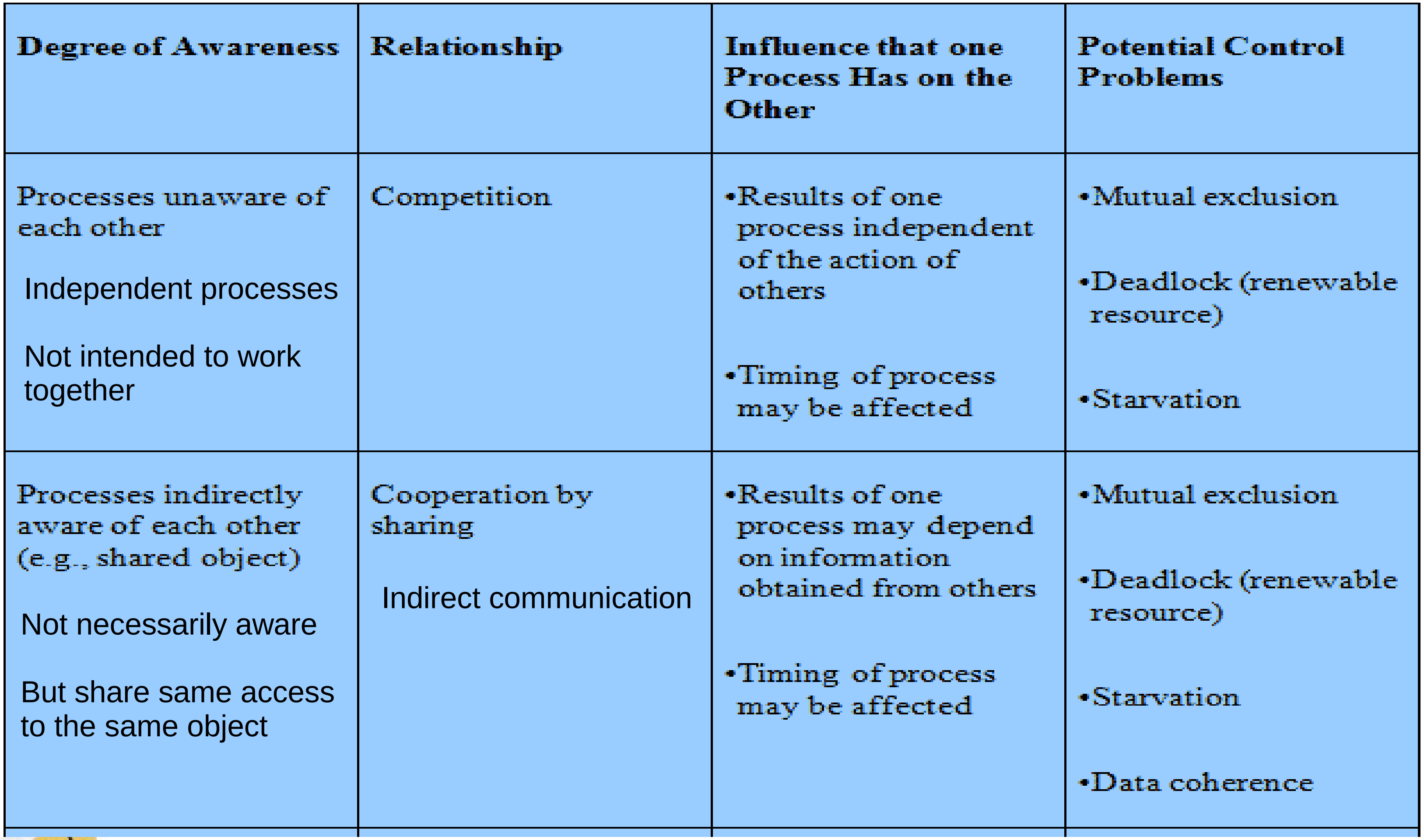
- OS must protect data and resources of each process against unintended interference by other processes

- The functioning of the OS & the output it produces must be independent of the speed of execution of other concurrent processes (not affected)

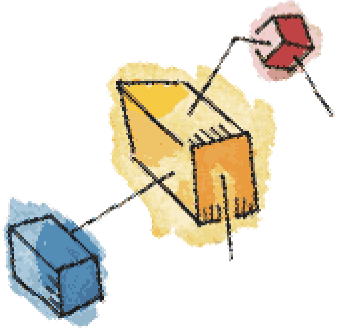


Process Interaction

Table 5.2 Process Interaction



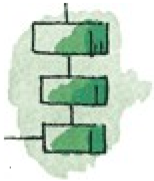
Degree of Awareness	Relationship	Influence that one Process Has on the Other	Potential Control Problems
Processes unaware of each other Independent processes Not intended to work together	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object) Not necessarily aware But share same access to the same object	Cooperation by sharing Indirect communication	•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

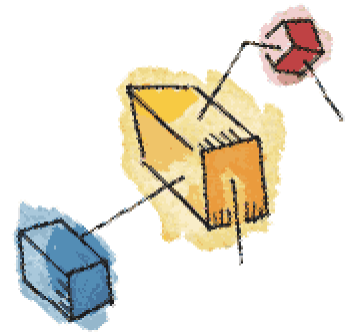


Process Interaction

Processes directly aware of each other (have communication primitives available to them) Jointly work together on some activities	Cooperation by communication Various processes participate in the common effort that link all of the processes Synchronise various activities	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Nothing is shared, so no Mutual exclusion

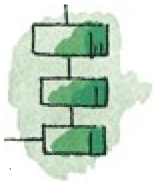




Competition among Processes for Resources

- Concurrent processes will come into conflict with each other when they are competing for the use of the same resources (I/O devices, memory, processor time, clock)
- Unaware of each other
- There is no exchange of information between competing processes

- Competition: One will get, another has to wait (slowed down)
- Extreme case: blocked process may never get access



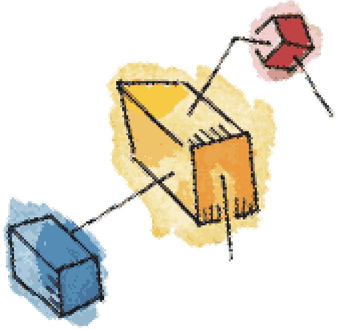
Potential Control Problem

➔Mutual Exclusion

- Critical resources
- Critical sections
  - E.g. Printer – line intersection

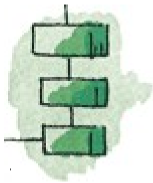
➔Deadlock

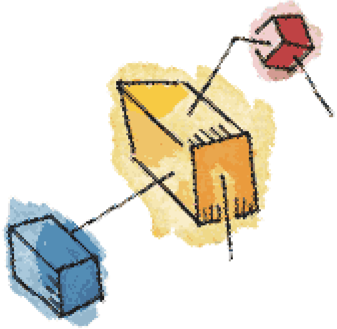
➔Starvation



Requirements for Mutual Exclusion

- Two or more processes require access to a single, non-sharable resources: e.g: printer (critical resources)
- Portion of program that uses it = critical section
- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- Process that acquiring access to critical section must not be delayed indefinitely: NO deadlock or starvation should occur





Requirements for Mutual Exclusion

- 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

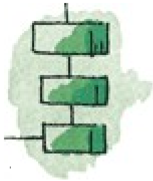
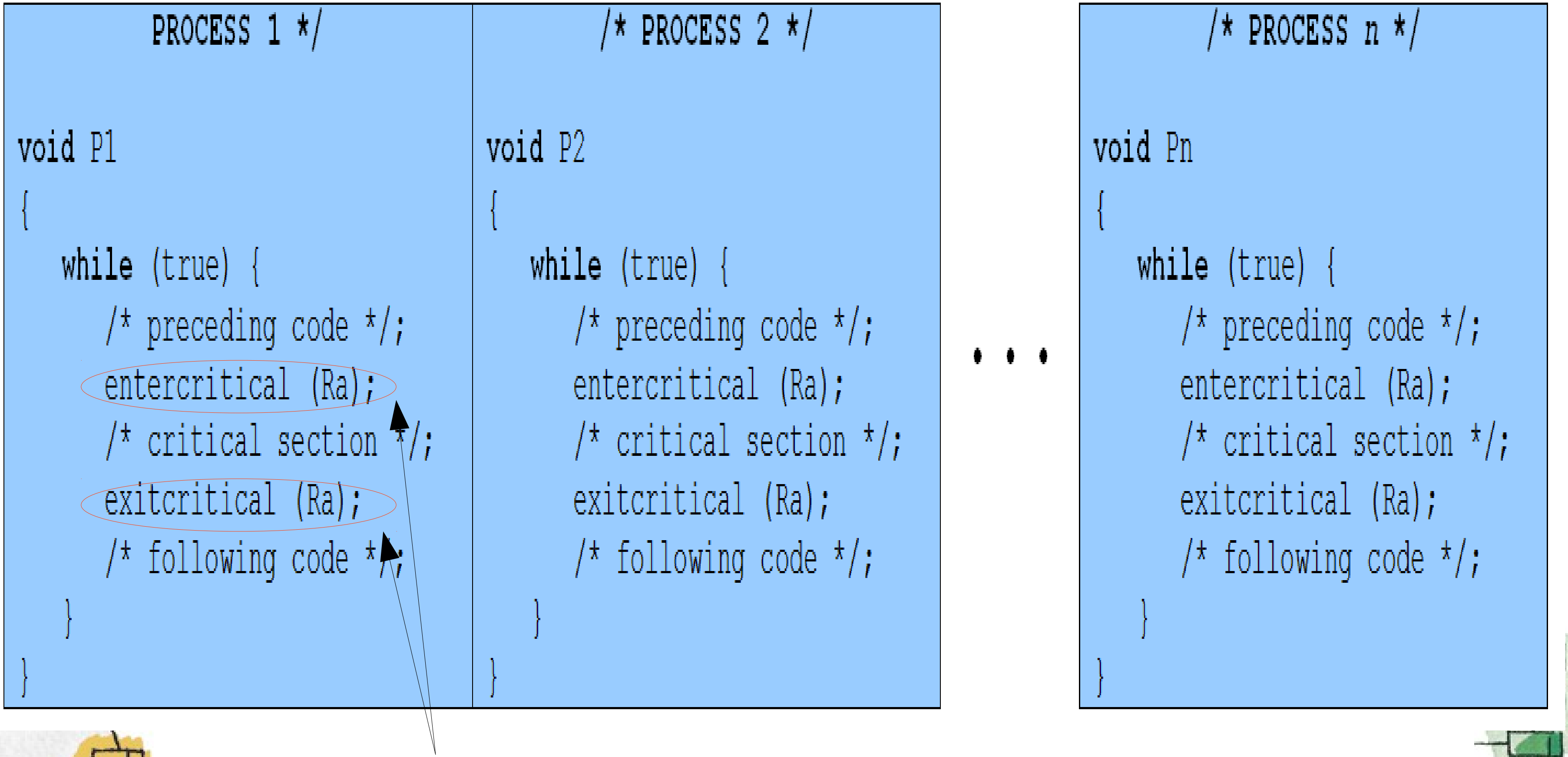


Illustration of Mutual Exclusion



```
        PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

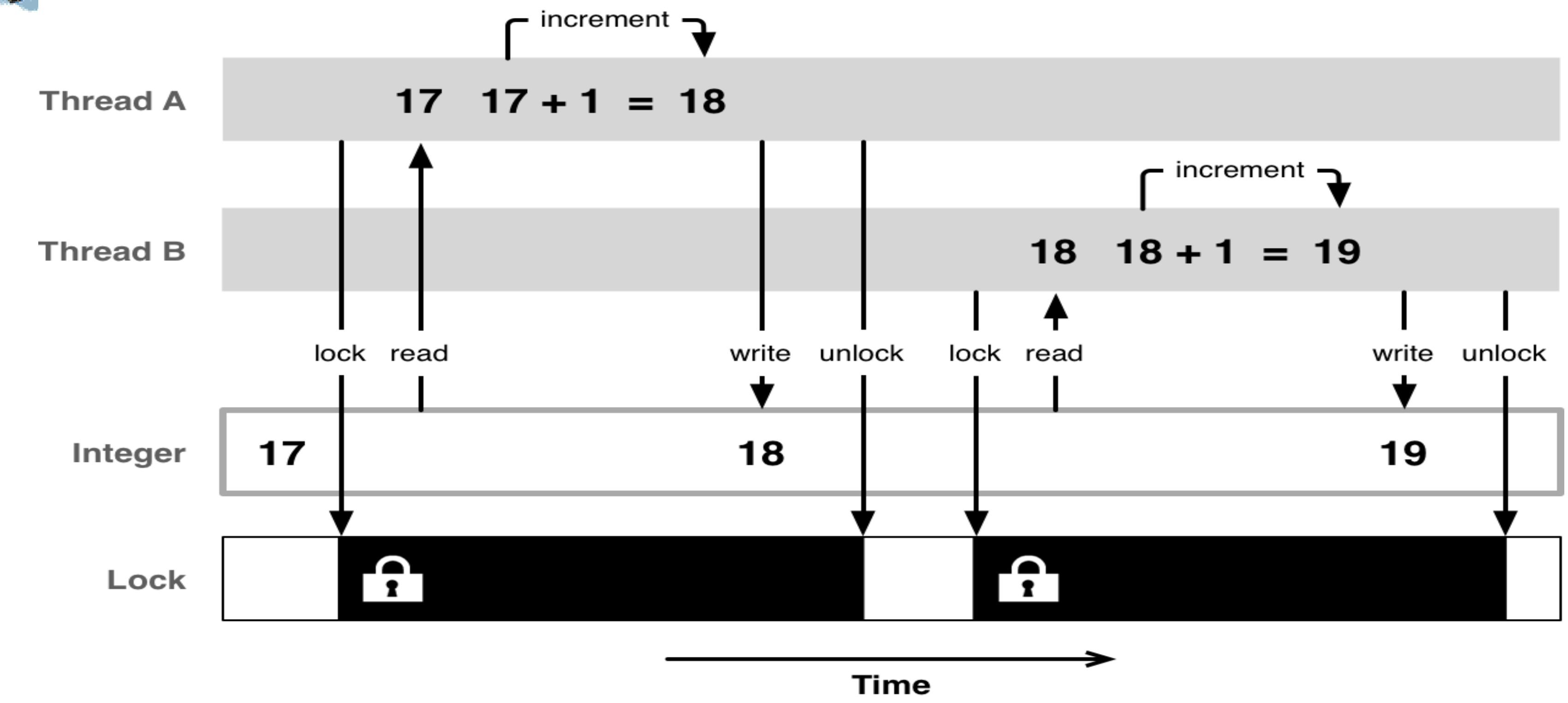
```
        /* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

• • •

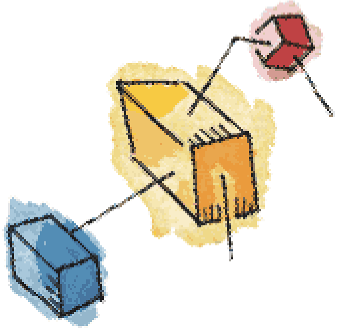
```
        /* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

Enforcement of Mutual Exclusion

Mutual Exclusion

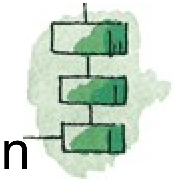


Mutual exclusive access means that only one thread at a time gets access to a certain resource. In order to ensure this, each thread that wants to access a resource first needs to acquire a mutex lock on it. Once it has finished its operation, it releases the lock, so that other threads get a chance to access it.

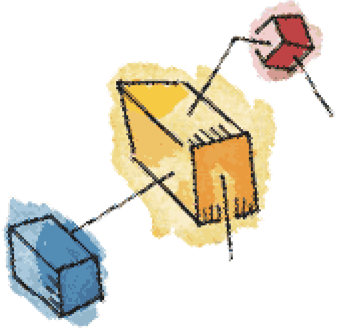


Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - In a uniprocessor system, concurrent processes cannot be overlapped, but can be interleaved
  - A process runs until it invokes an operating system service or until it is interrupted
  - To enable mutual exclusion, NO INTERRUPT can occur.
  - Disabling interrupts in critical section guarantees mutual exclusion
  - However, processor is limited in its ability to interleave programs, degrade performance
  - Problem: Will not work in multiprocessor architecture



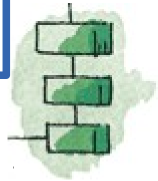
Disable interrupt does not guarantee mutual exclusion



Mutual Exclusion: Hardware Support

- Special Machine Instruction:Compare&Swap

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



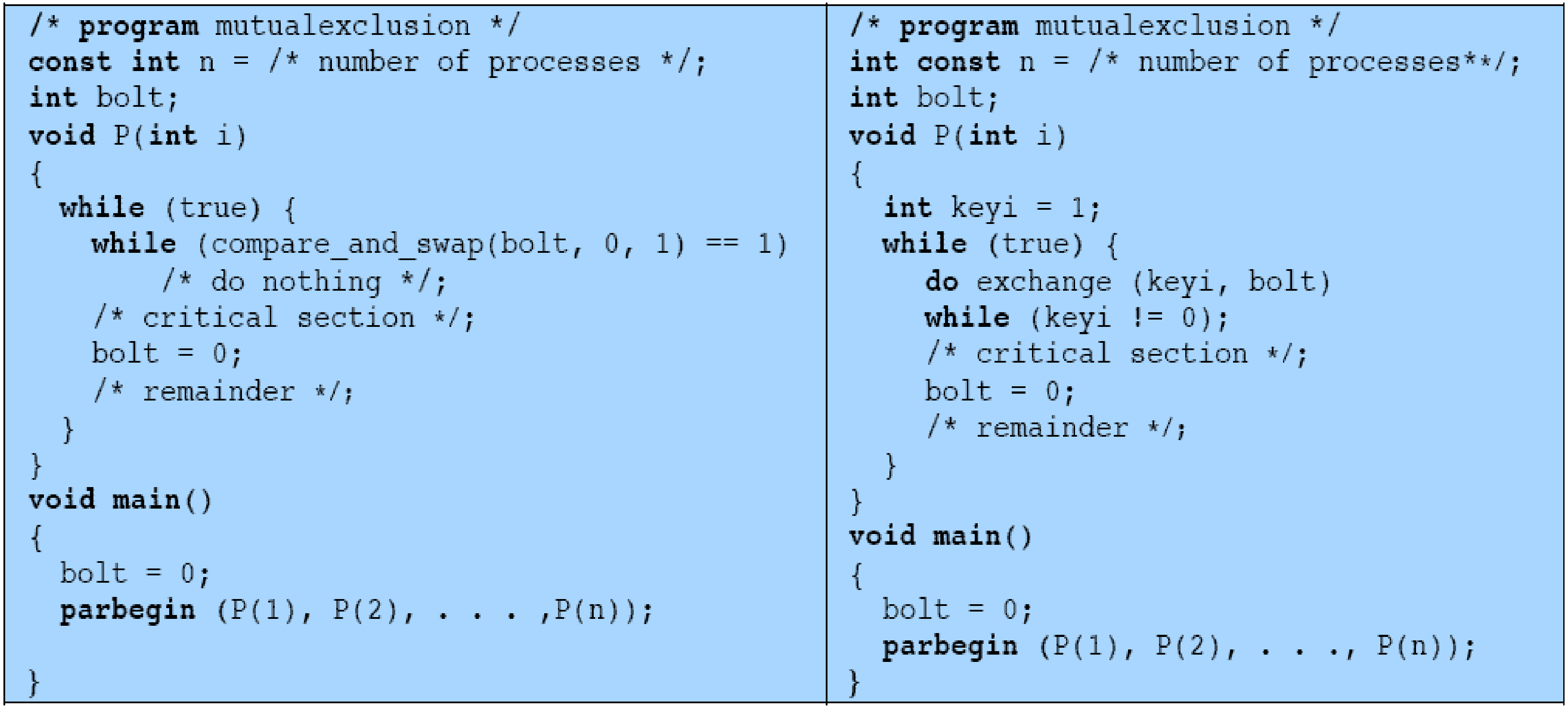
During execution of instruction, access to the memory is blocked for any other instruction referencing that location (atomic)

Mutual Exclusion: Hardware Support

- Special Machine Instruction: Exchange instruction

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

Mutual Exclusion



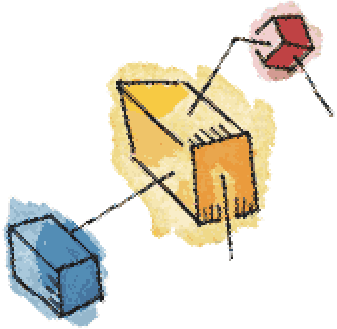
```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(bolt, 0, 1) == 1)
         /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1),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

}
```

(a) Compare and swap instruction

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

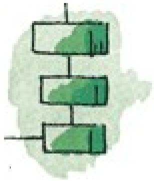
(b) Exchange instruction

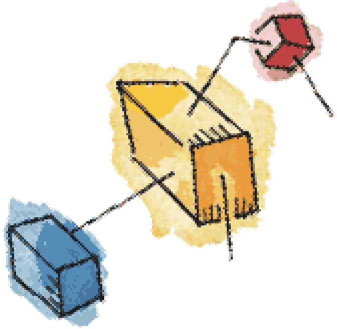
Figure 5.2 Hardware Support for Mutual Exclusion



Mutual Exclusion (Machine-Instruction)

- 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

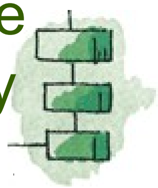




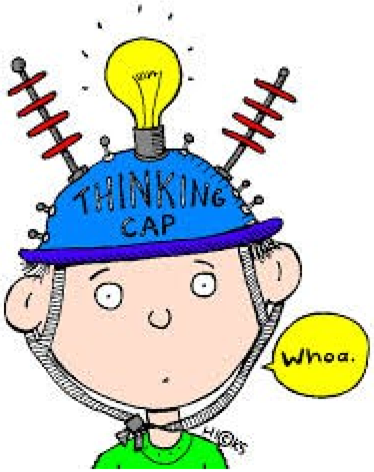
Mutual Exclusion Machine-Instruction

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible
    - if more than one processes is waiting, the selection is arbitrary (some can be denied access).
  - Deadlock is still possible

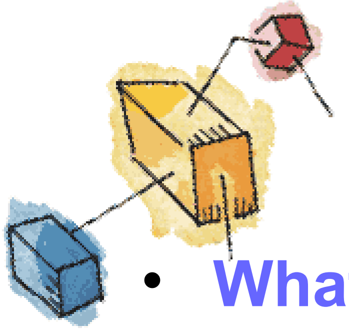
**Busy waiting : All other processes attempting to enter the critical sections are put into busy waiting mode where they can do nothing until the permission is granted



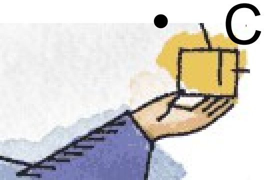
Because of the drawback of both the software and hardware solutions, we need to find other mechanisms to implement concurrency

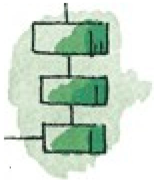


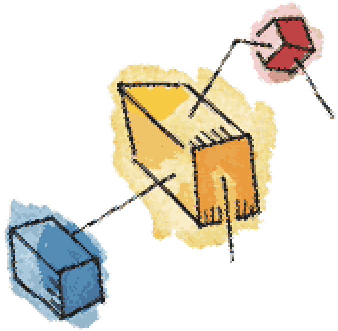
Semaphores



- **What is it?**
  - Programming language mechanism to provide concurrency
  - An integer value used for signaling among processes
  - Protected signal that can restrict access to the shared resources
    - Stop at specific place until it receive a specific signal
  - simply a way to limit the number of consumers for a specific resource
- If a process is waiting for a signal, it is suspended until that signal is sent
- Can be used to restrict access to a certain resource to a maximum (but variable) number of processes.
- Commonly used to control access to files and shared memory

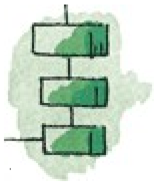


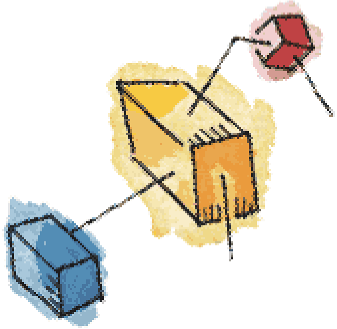




Semaphores

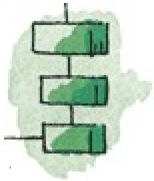
- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value
- useful tool in the prevention of race conditions

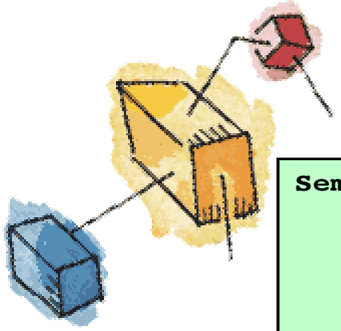




Example

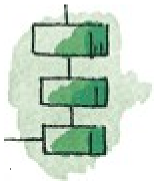
- Think of semaphores as people in a restaurant. There are a dedicated number of people that are allowed in the restaurant at once (e.g. 50).
- Initially, semaphores (s) is set to 50.
- As each person arrives at restaurant, s= s-1.
- When restaurant is full, s=0
- If the restaurant is full, no one is allowed to enter, but as soon as one person leaves another person might enter . Here s= s+1
- It's simply a way to limit the number of consumers for a specific resource.



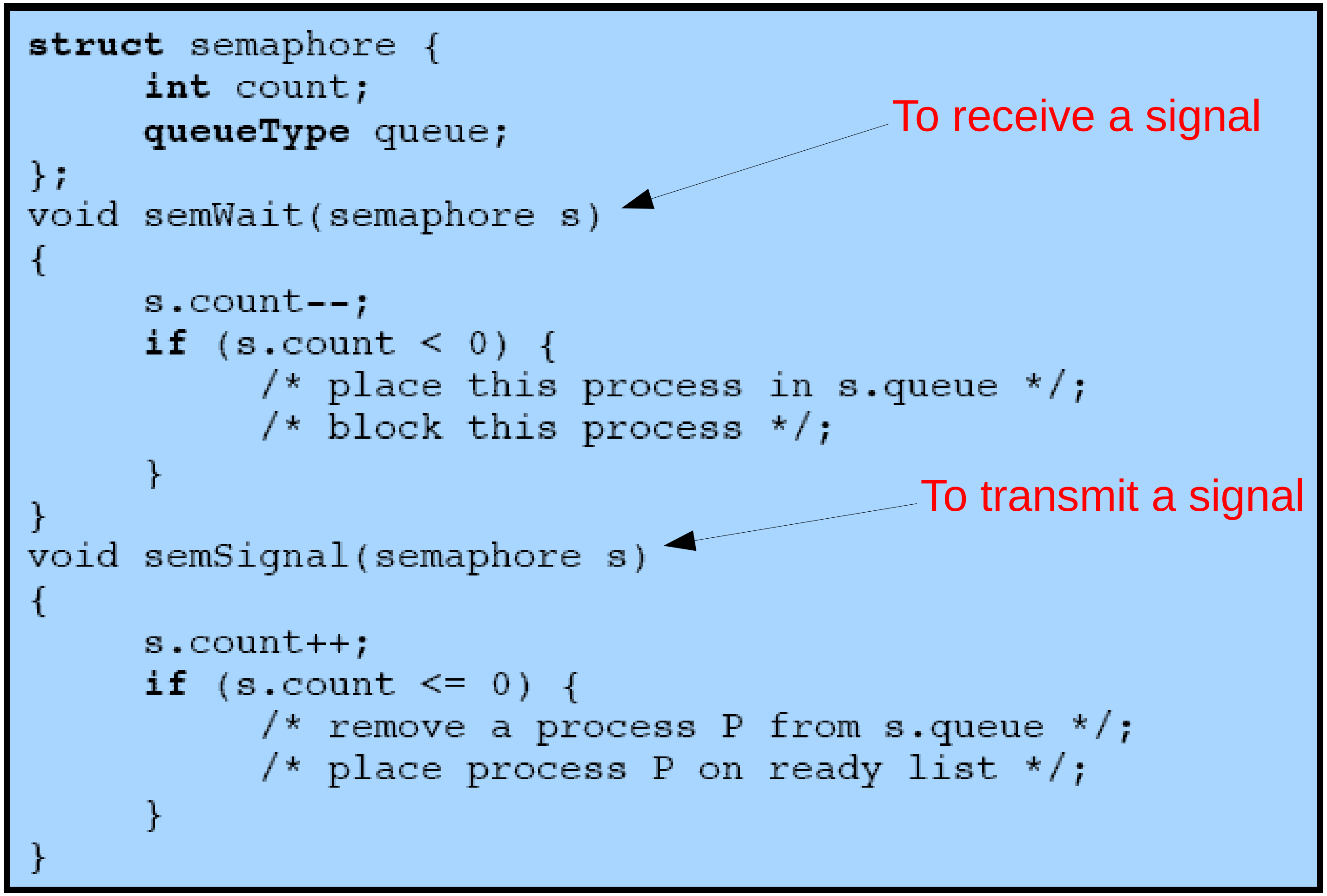


Common concurrency mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.



Semaphore Primitives



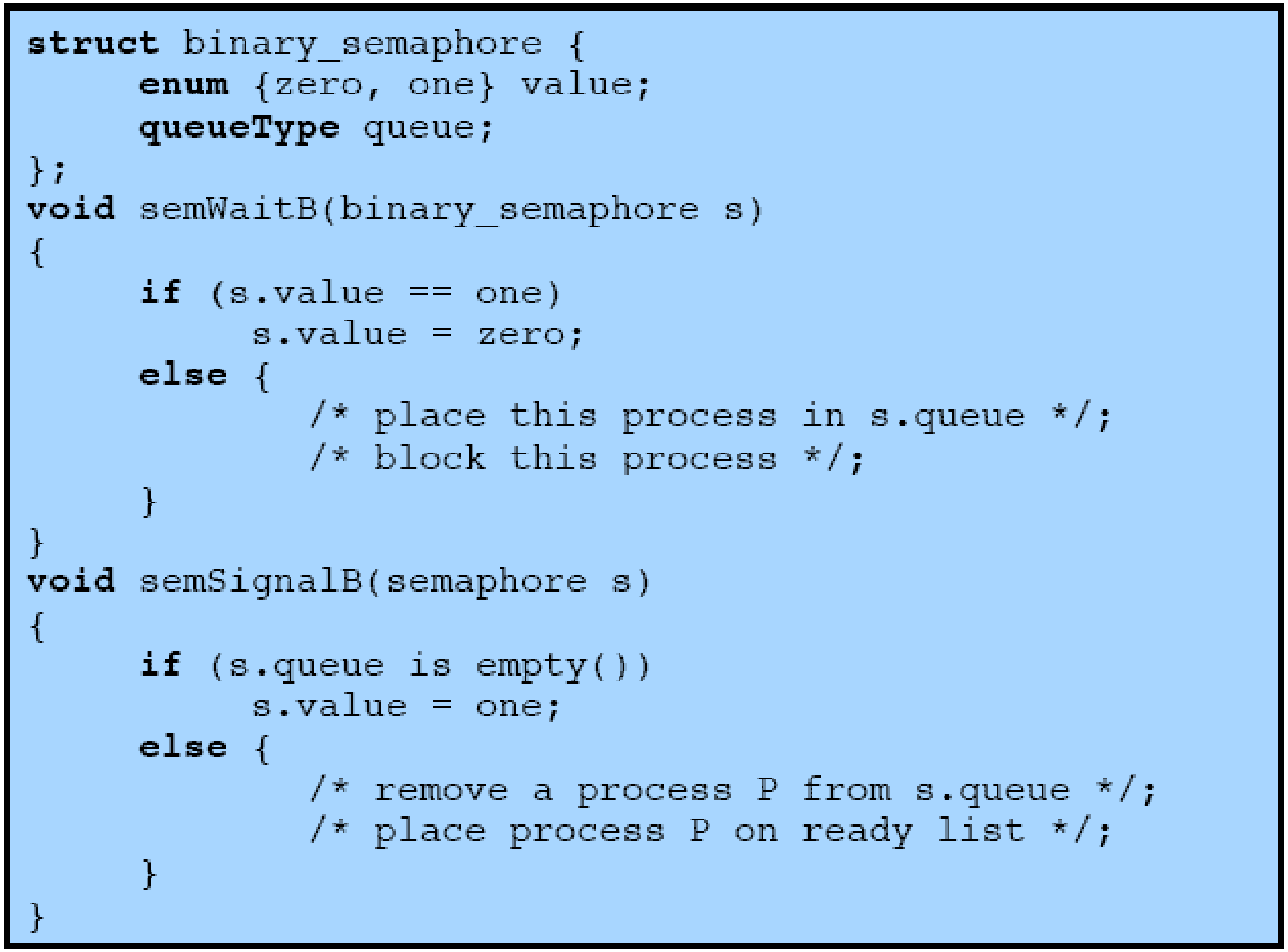
```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
     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

To receive a signal

To transmit a signal

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

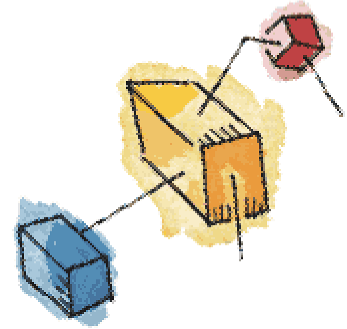


```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore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

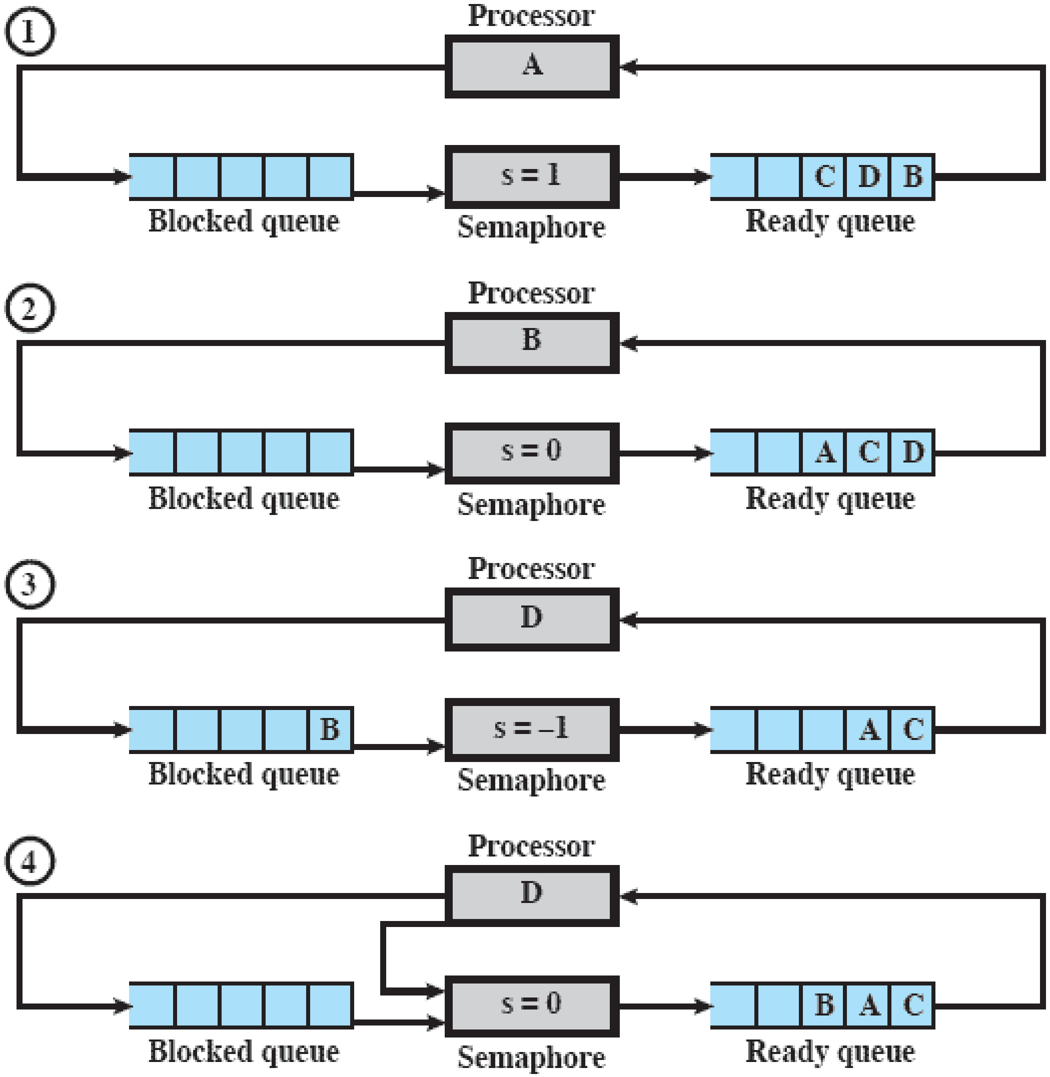
Figure 5.4 A Definition of Binary Semaphore Primitives

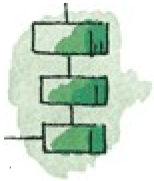
Semaphores queue

- A queue is used to hold processes waiting on the semaphores
- Two types:
  - **Strong semaphores**
    - FIFO
  - **Weak semaphores**
    - No specific order



Example of Semaphore Mechanism





Example of Semaphore Mechanism

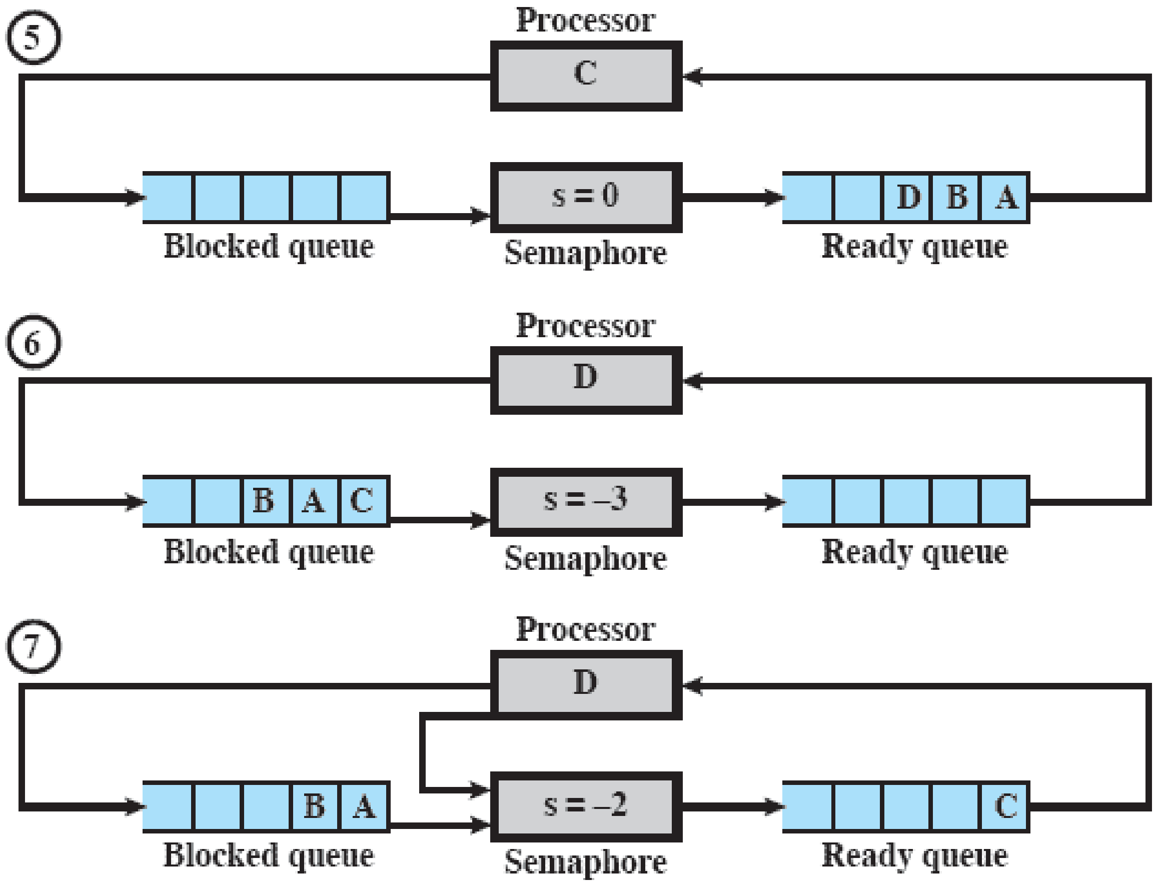
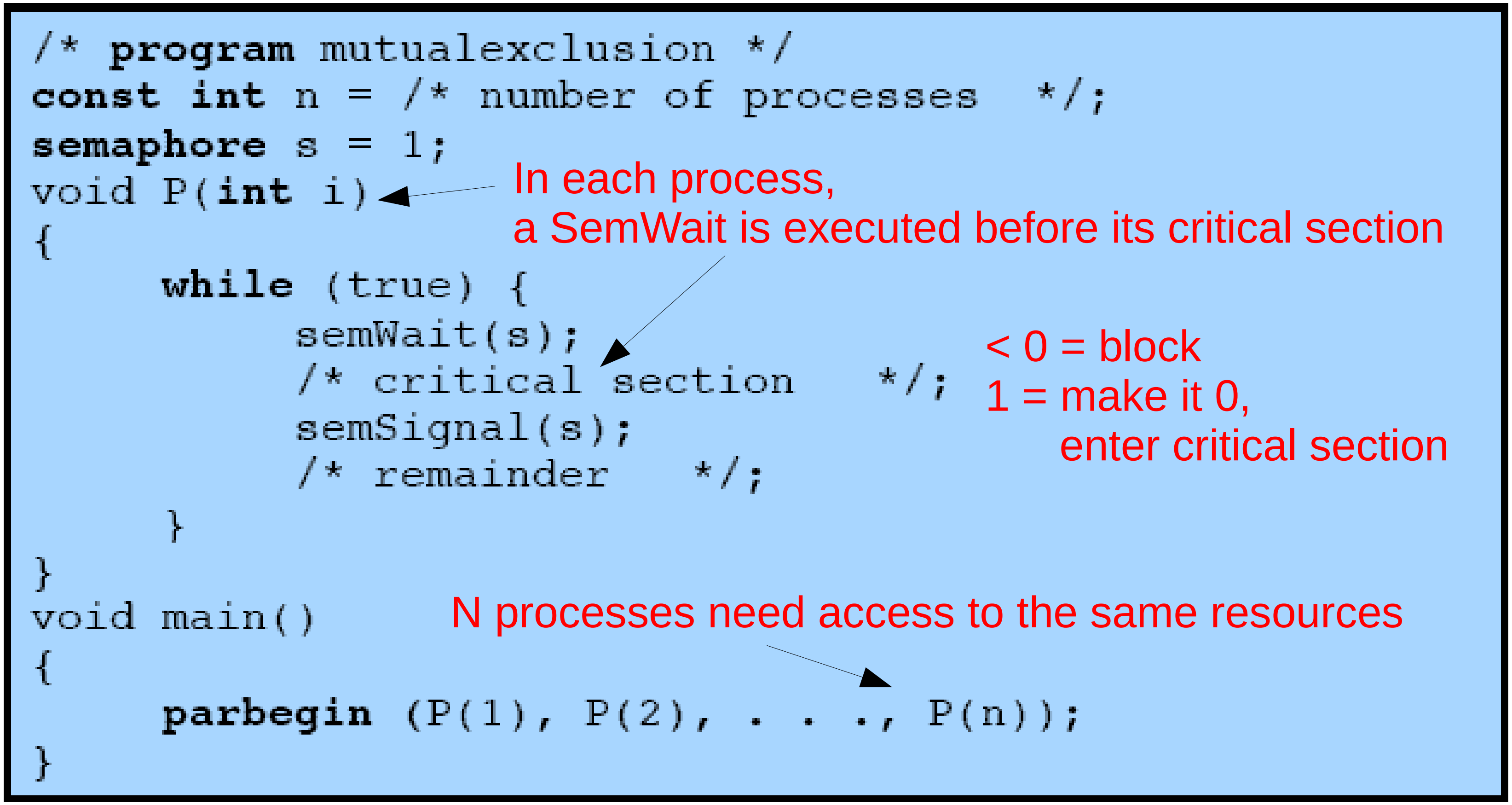


Figure 5.5 Example of Semaphore Mechanism

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));
}
```

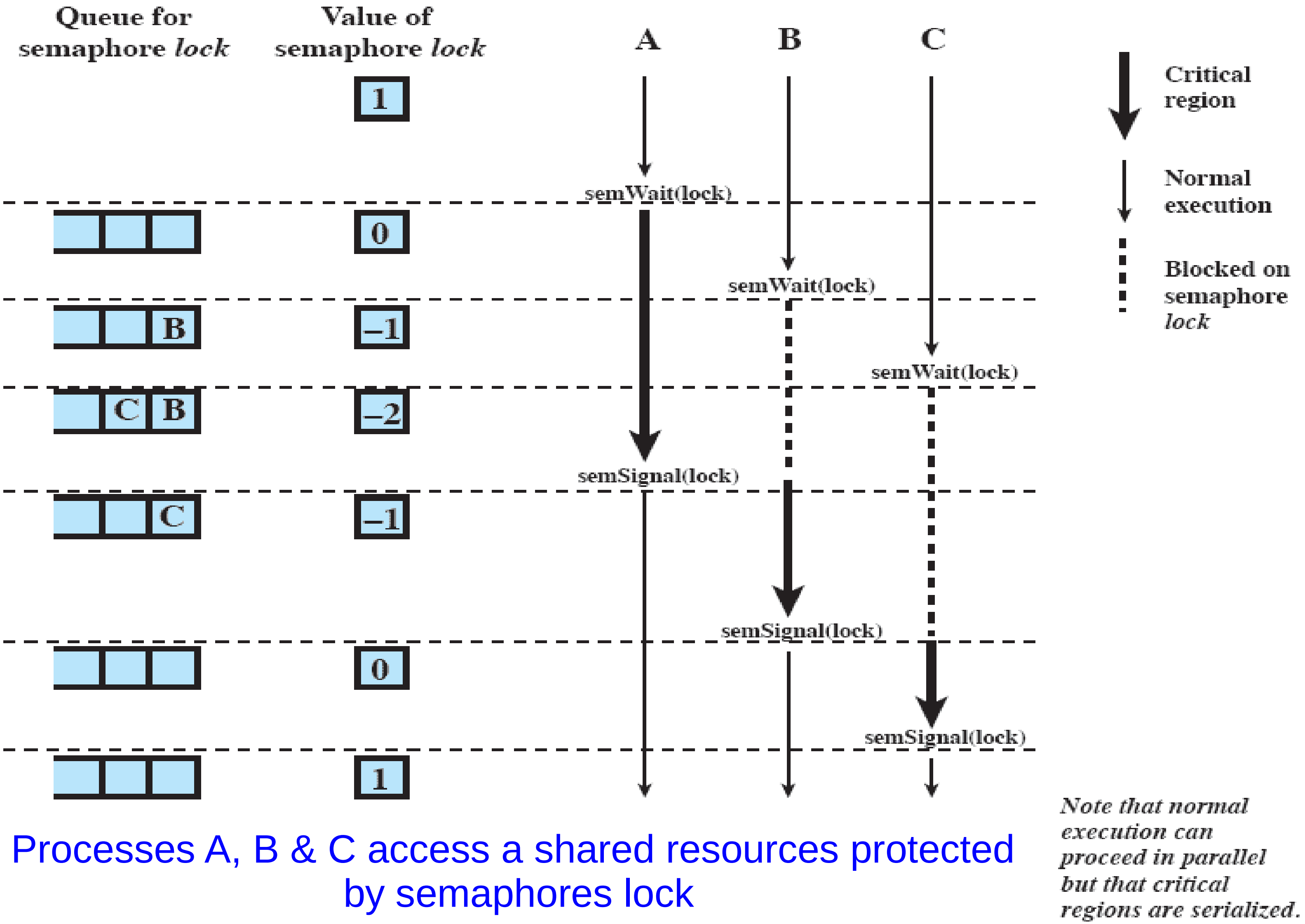
In each process,
a SemWait is executed before its critical section

< 0 = block
1 = make it 0,
enter critical section

N processes need access to the same resources

Figure 5.6 Mutual Exclusion Using Semaphores

Processes Using Semaphore

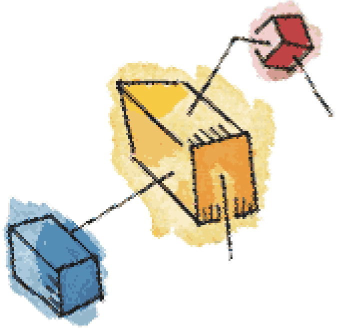


Note that normal execution can proceed in parallel but that critical regions are serialized.

Processes A, B & C access a shared resources protected by semaphores lock

Producer/Consumer Problem

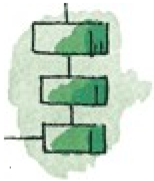
- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at one time
- Producer can’t add data into full buffer and consumer can’t remove data from empty buffer



Producer

- producer:

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

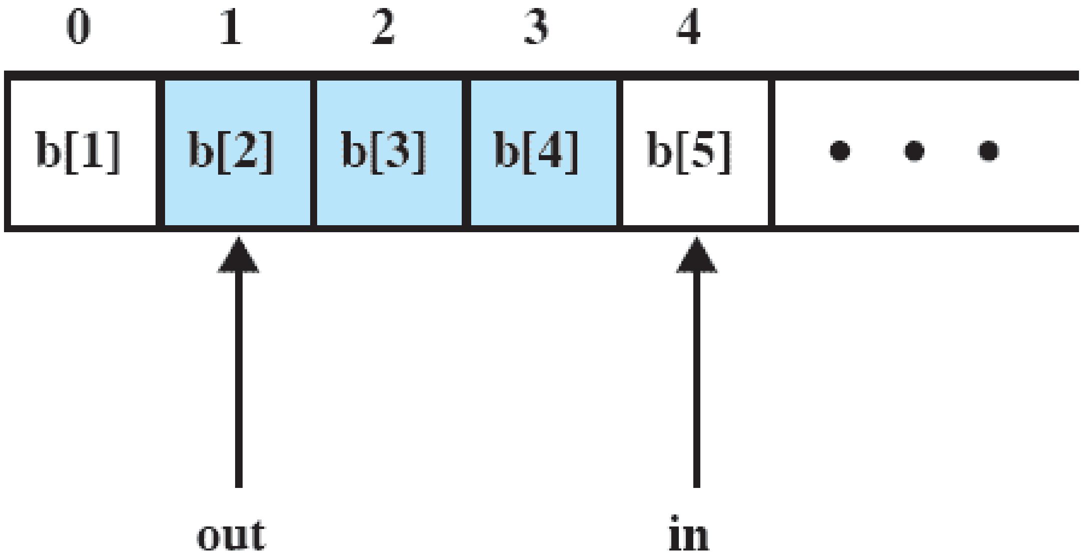


Consumer

- consumer:

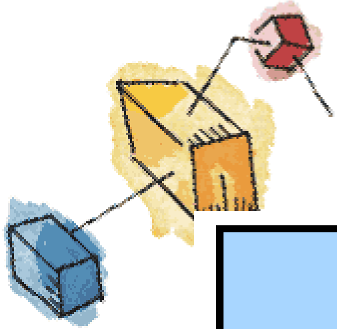
```
while (true) {
   while (in <= out)
       /*do  nothing */;
   w = b[out];
   out++;
   /* consume item w */
}
```

Buffer

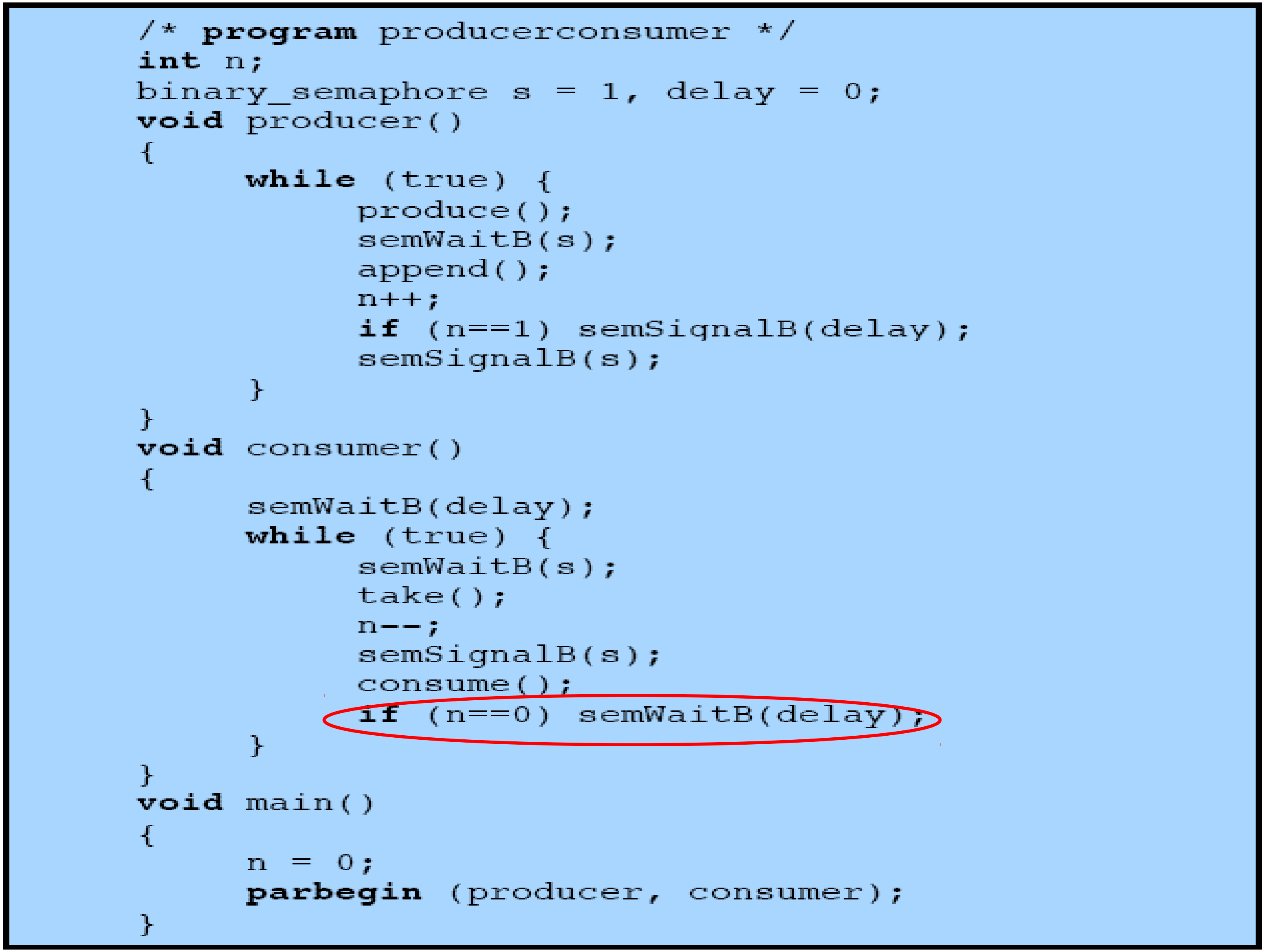


Note: shaded area indicates portion of buffer that is occupied

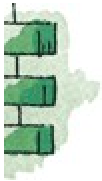
Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

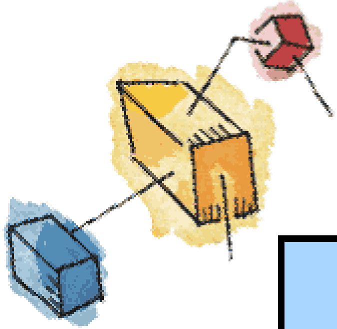


Incorrect Solution

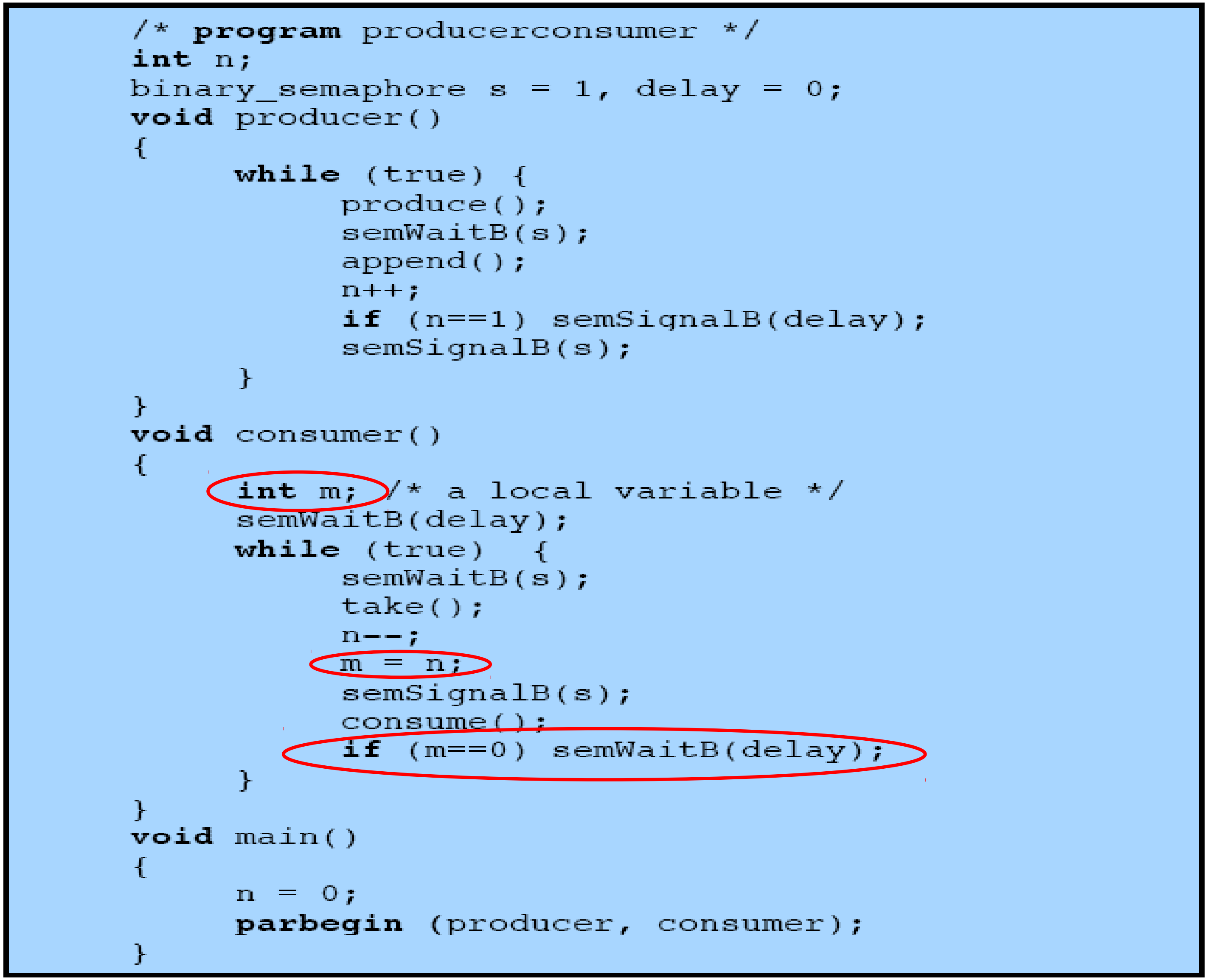


```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);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

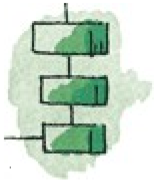




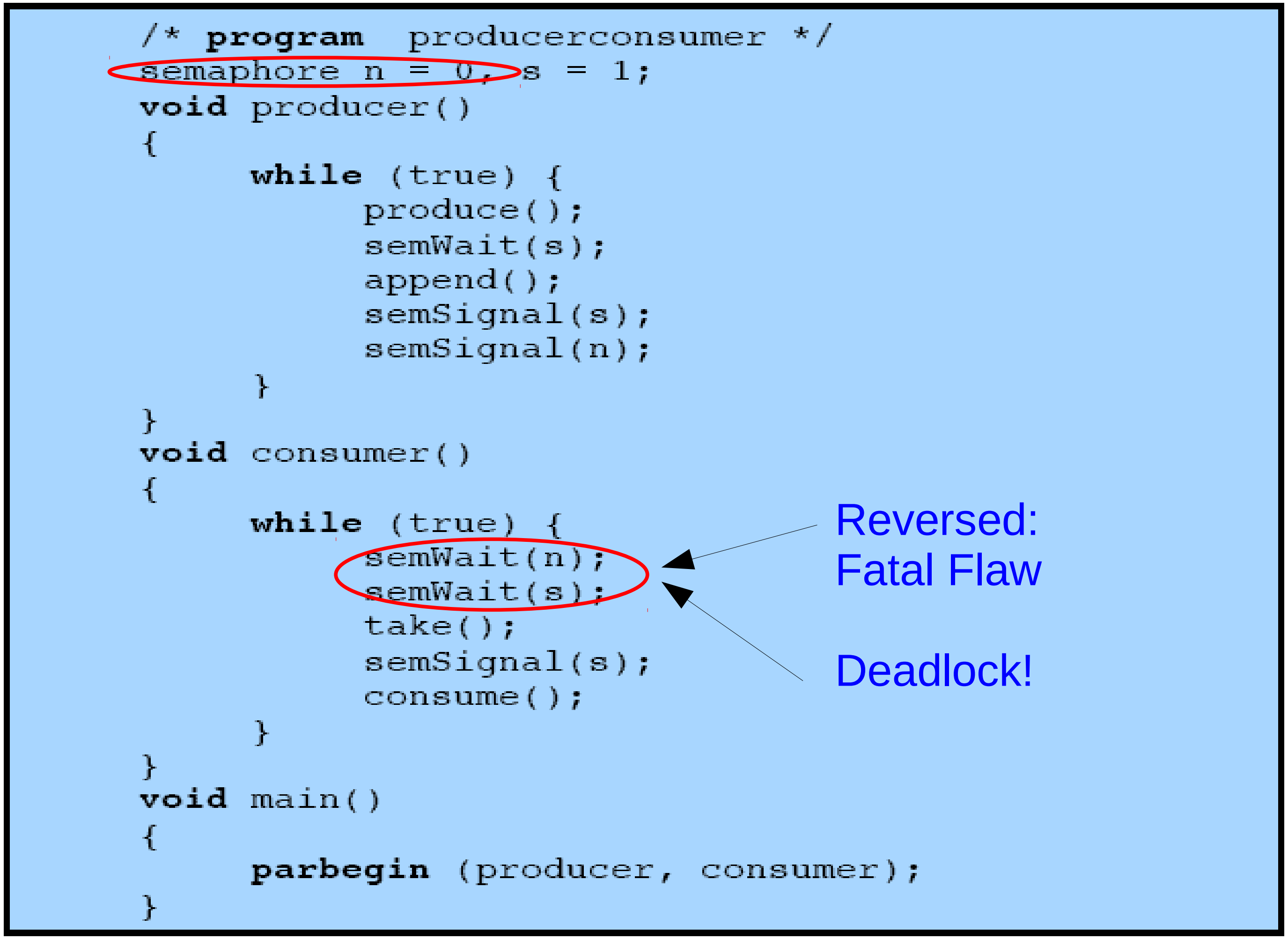
Correct Solution



```
/* 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);
}
```



Semaphores

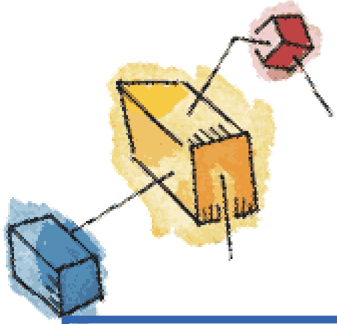


```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
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

Reversed:
Fatal Flaw

Deadlock!

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



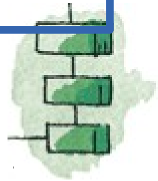
Producer with Circular Buffer

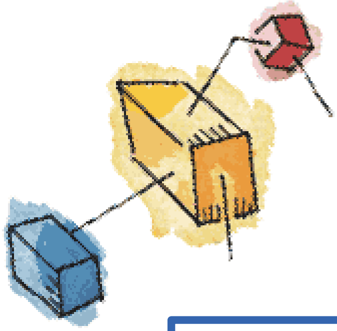
- producer:

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

In = 0
Out = 0

If no space left (full),



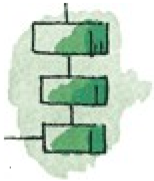


Consumer with Circular Buffer

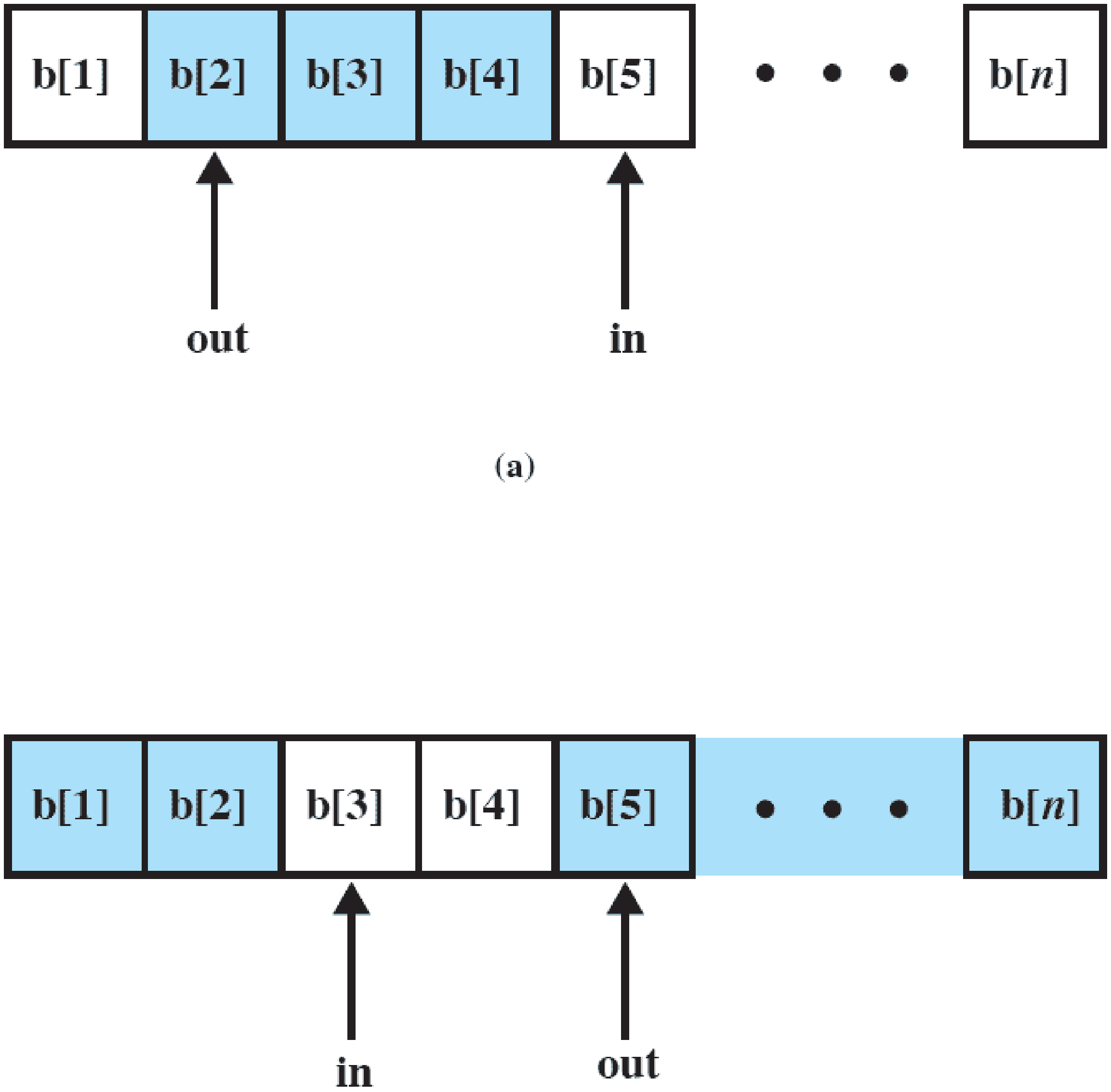
- consumer:

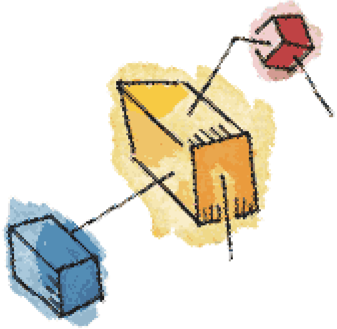
```
while (true) {
   while (in == out)
      /* do nothing */;
   w = b[out];
   out = (out + 1) % n;
   /* consume item w */
}
```

If buffer is empty



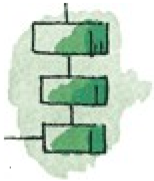
Circular Buffer

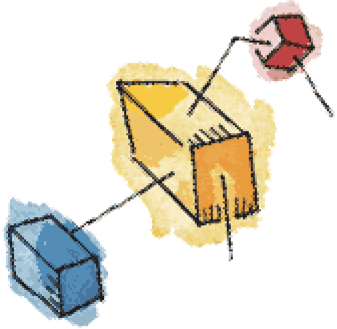




Monitors

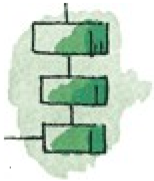
- Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time



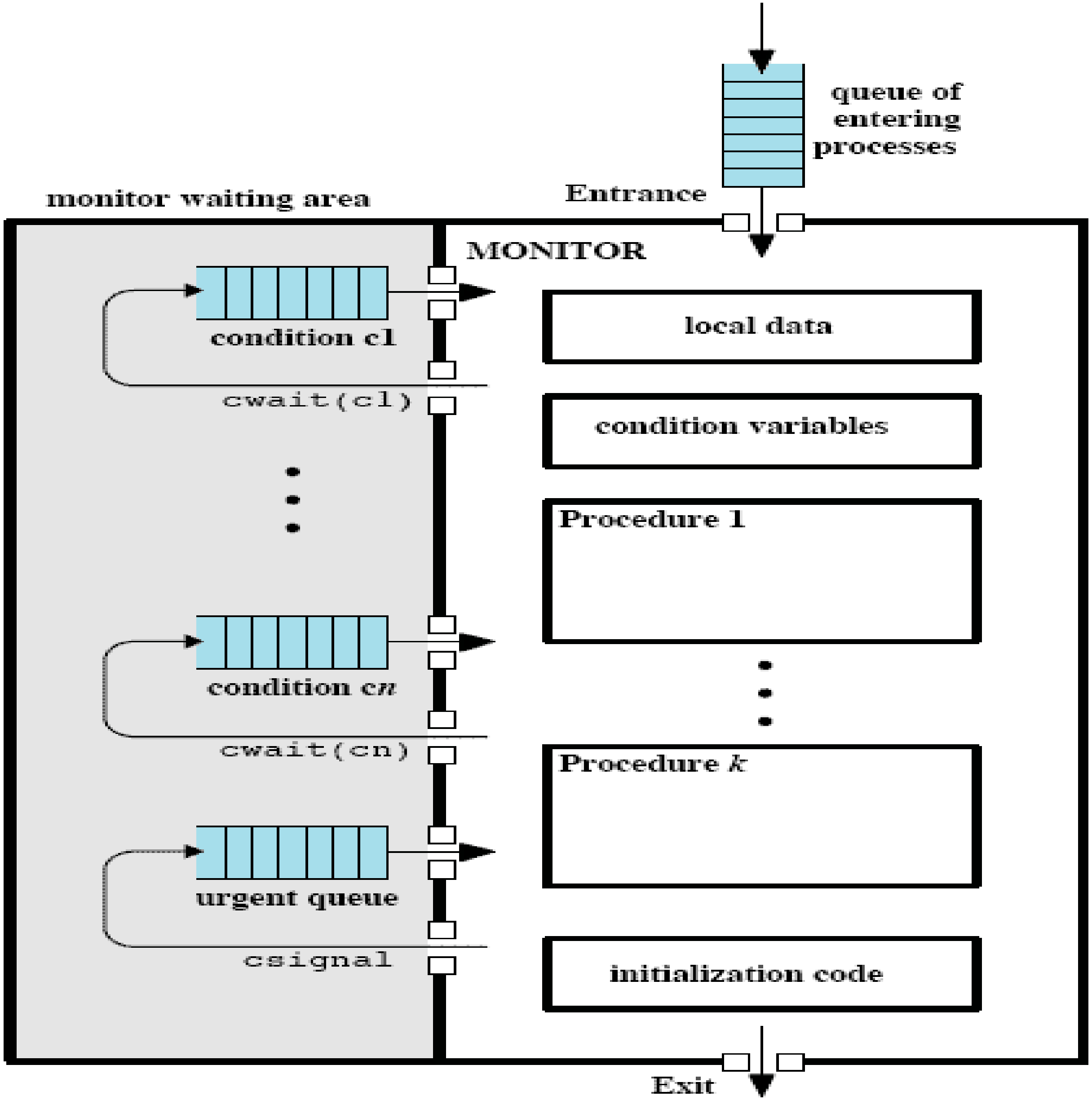


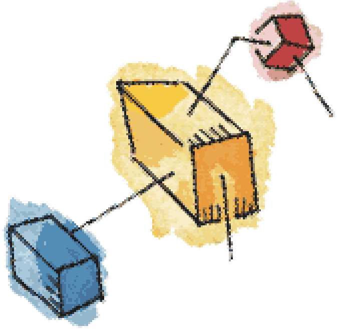
Monitors

- Support synchronization by the use of condition variables:
  - cwait(c): suspend execution of the calling process on condition c.
  - csignal(c): resume execution of some process blocked after a cwait on the same condition.

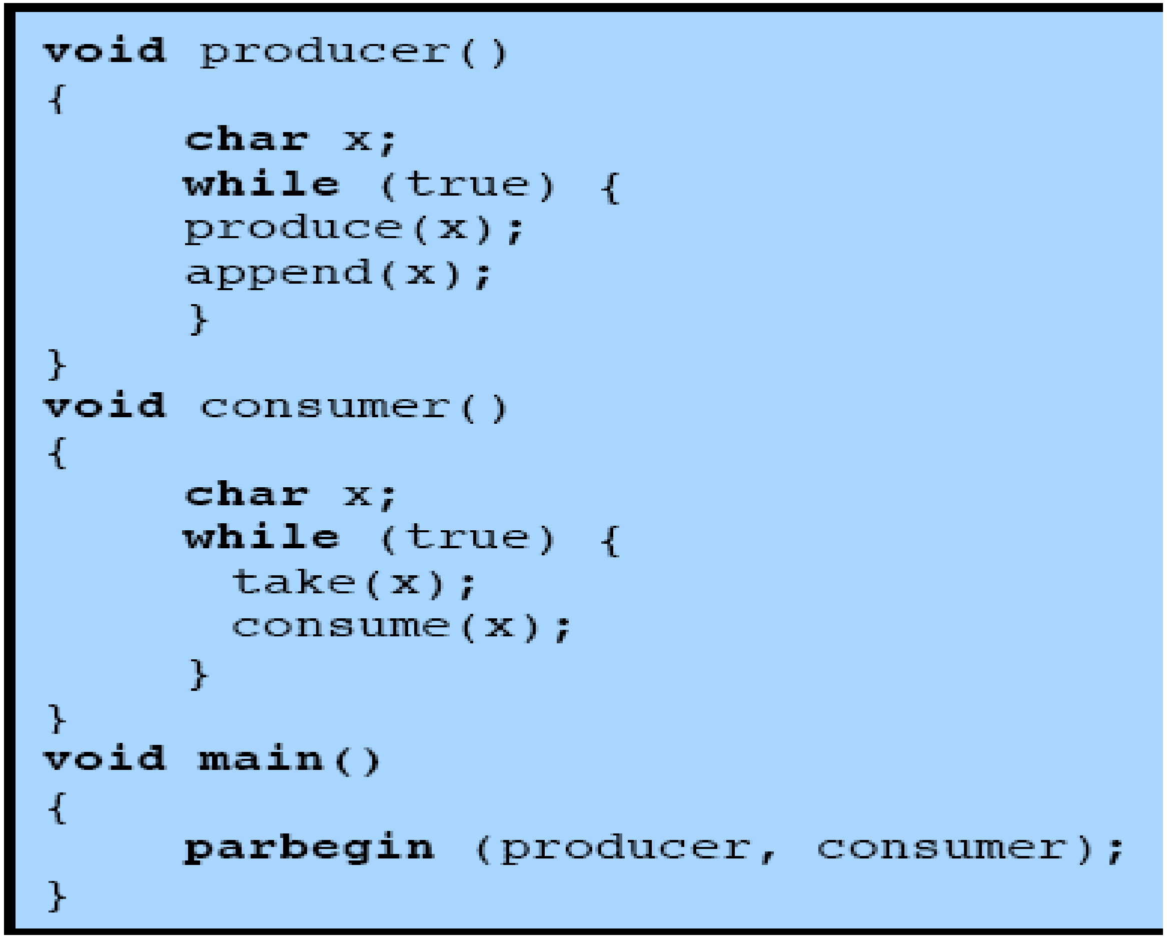


Structure of a Monitor

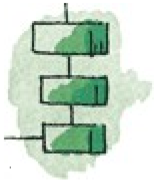


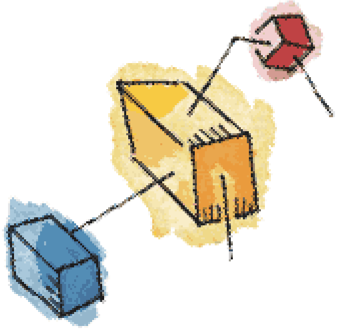


Solution Using Monitor



```
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);
}
```



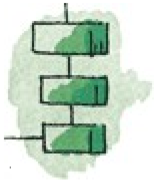


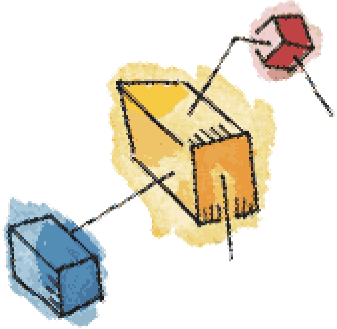
Message Passing

- Two requirements:
  - Synchronization (Enforce mutual exclusion)
  - Communication (Exchange information)

Key: Message Passing!

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

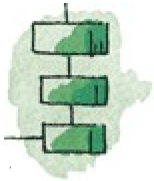


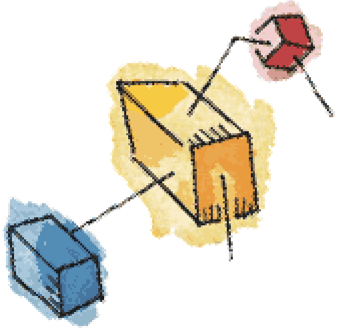


Synchronization

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

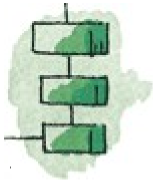
- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a rendezvous

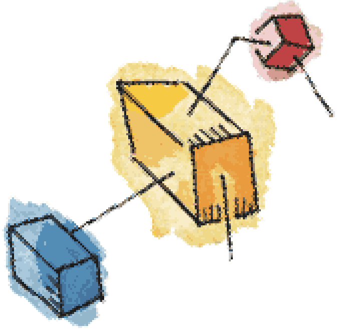




Synchronization

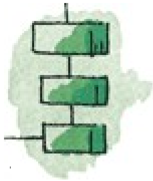
- Nonblocking send, blocking receive
  - – Sender continues on
  - – Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - – Neither party is required to wait

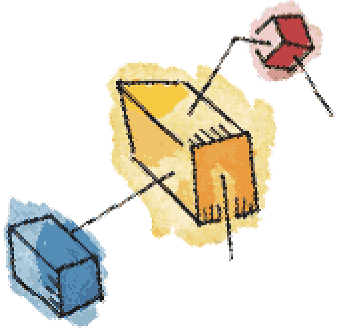




Addressing

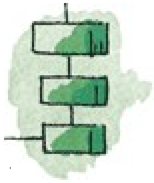
- Direct addressing
  - Send primitive includes a specific identifier of the destination process
  - Receive primitive could know ahead of time which process a message is expected
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

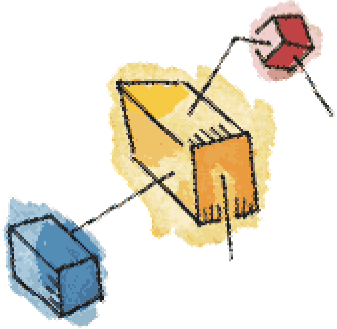




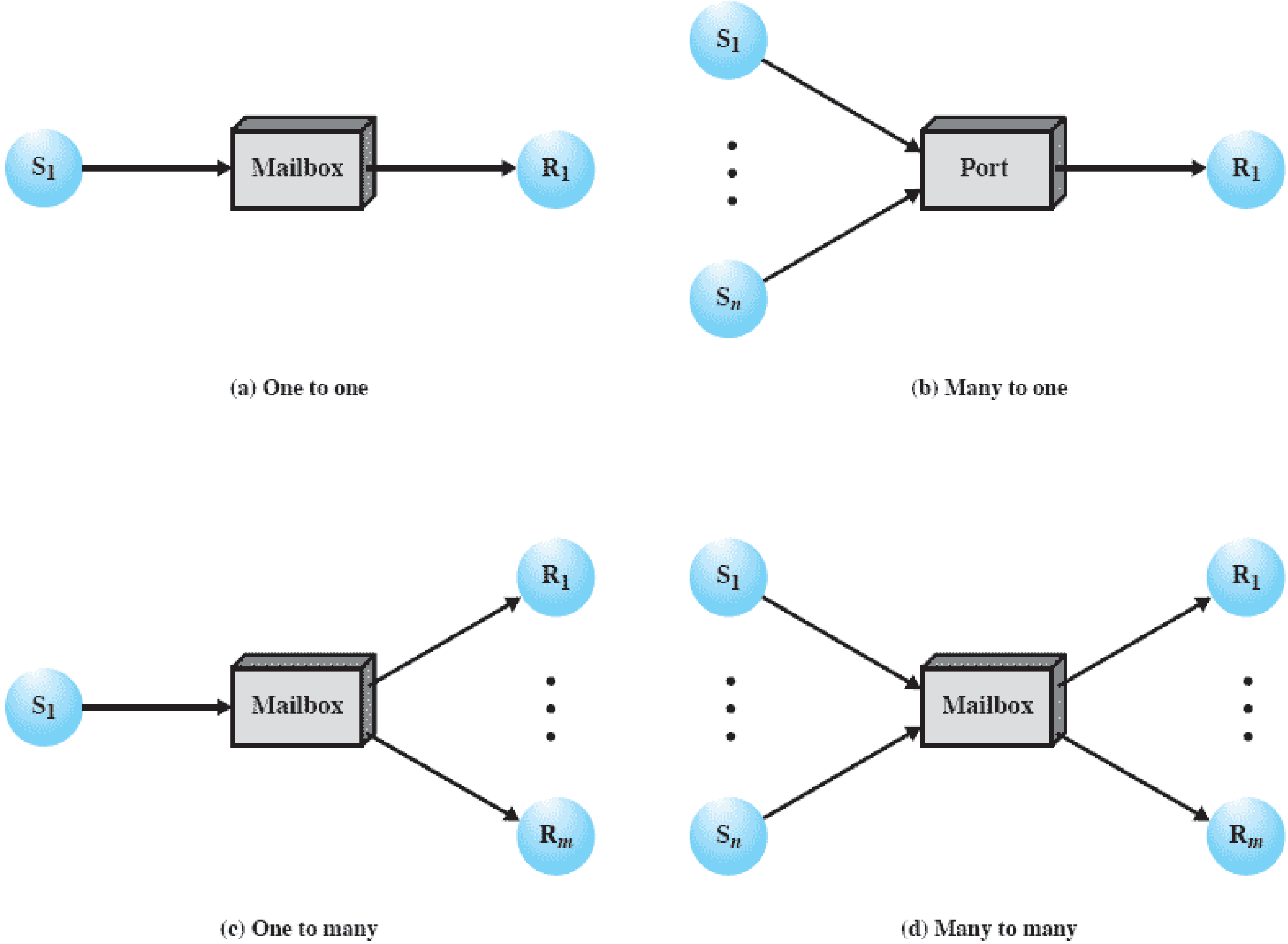
Addressing

- 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





Indirect Process Communication



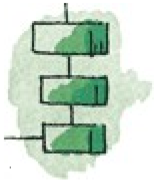
(a) One to one

(b) Many to one

(c) One to many

(d) Many to many

Figure 5.18 Indirect Process Communication



General Message Format

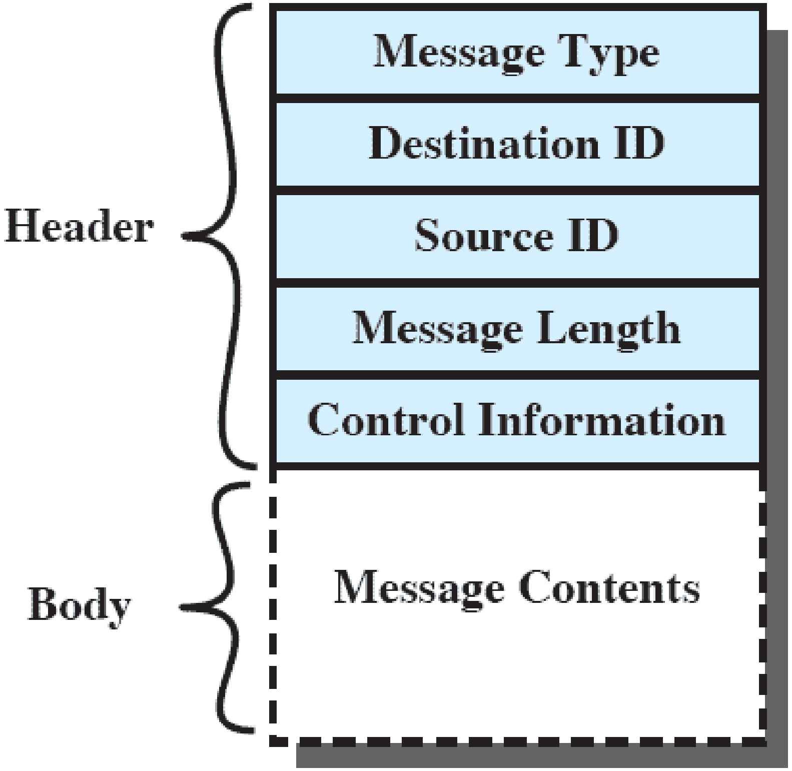
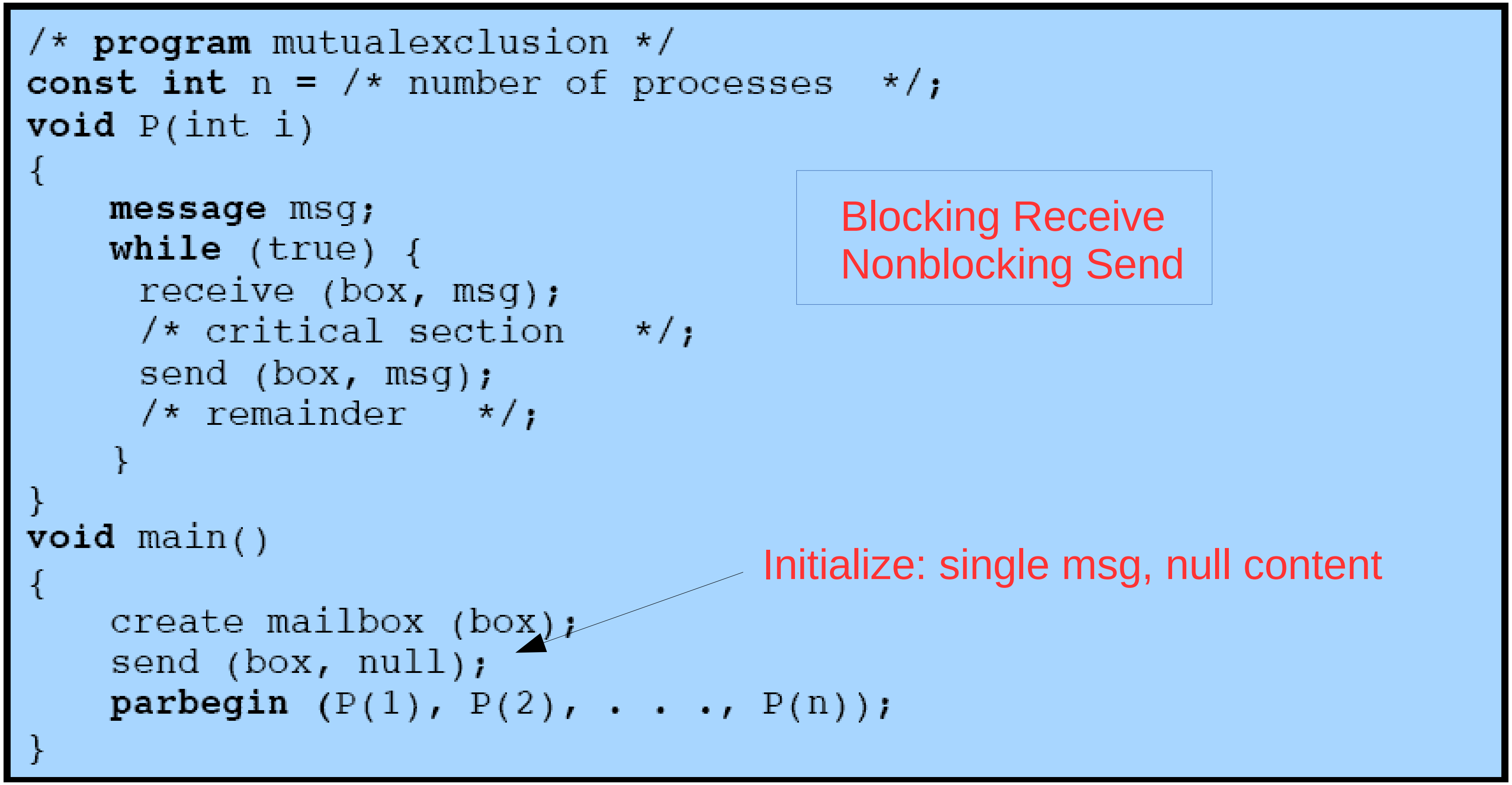


Figure 5.19 General Message Format

Mutual Exclusion Using Messages



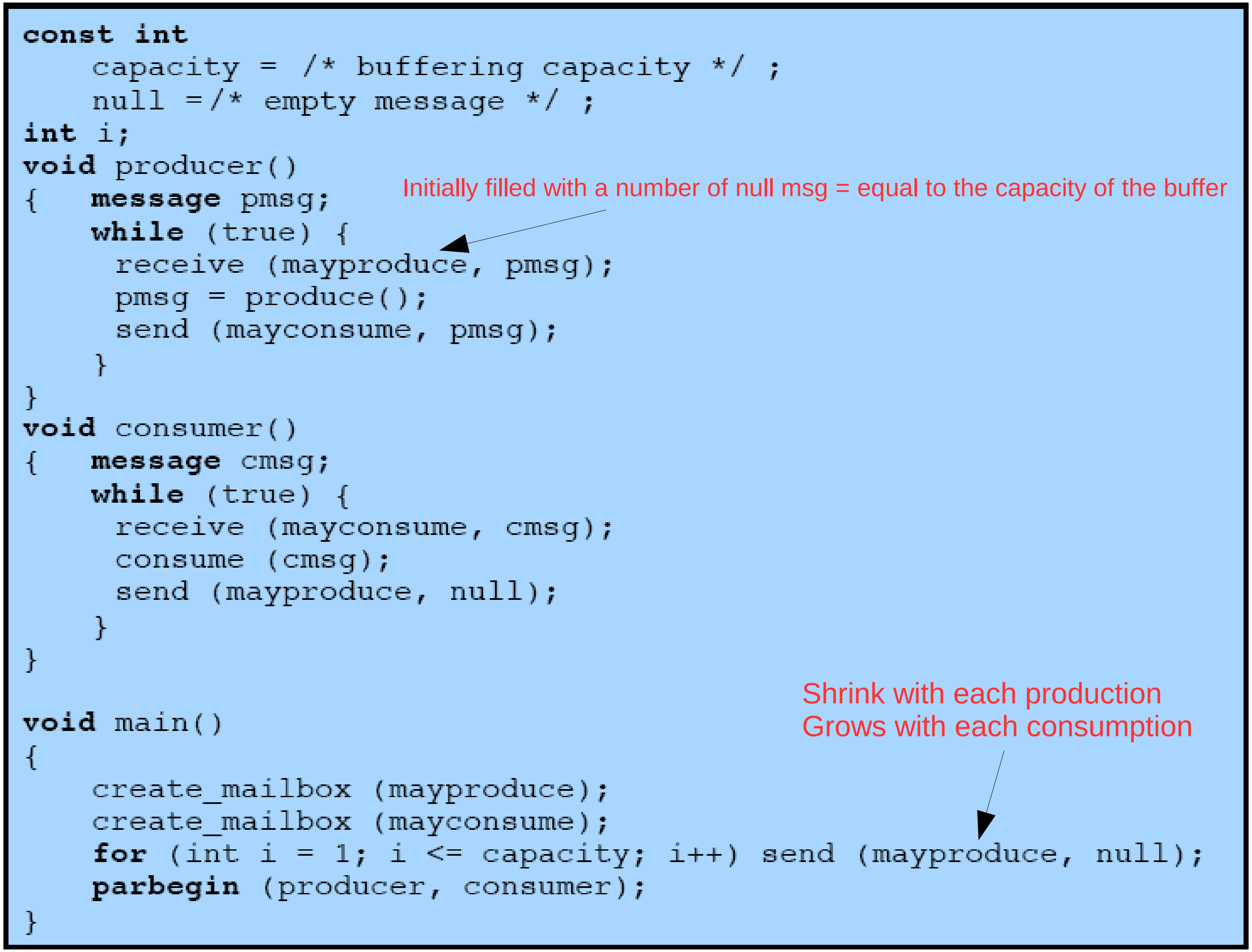
```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
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

Blocking Receive
Nonblocking Send

Initialize: single msg, null content

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);
}
```

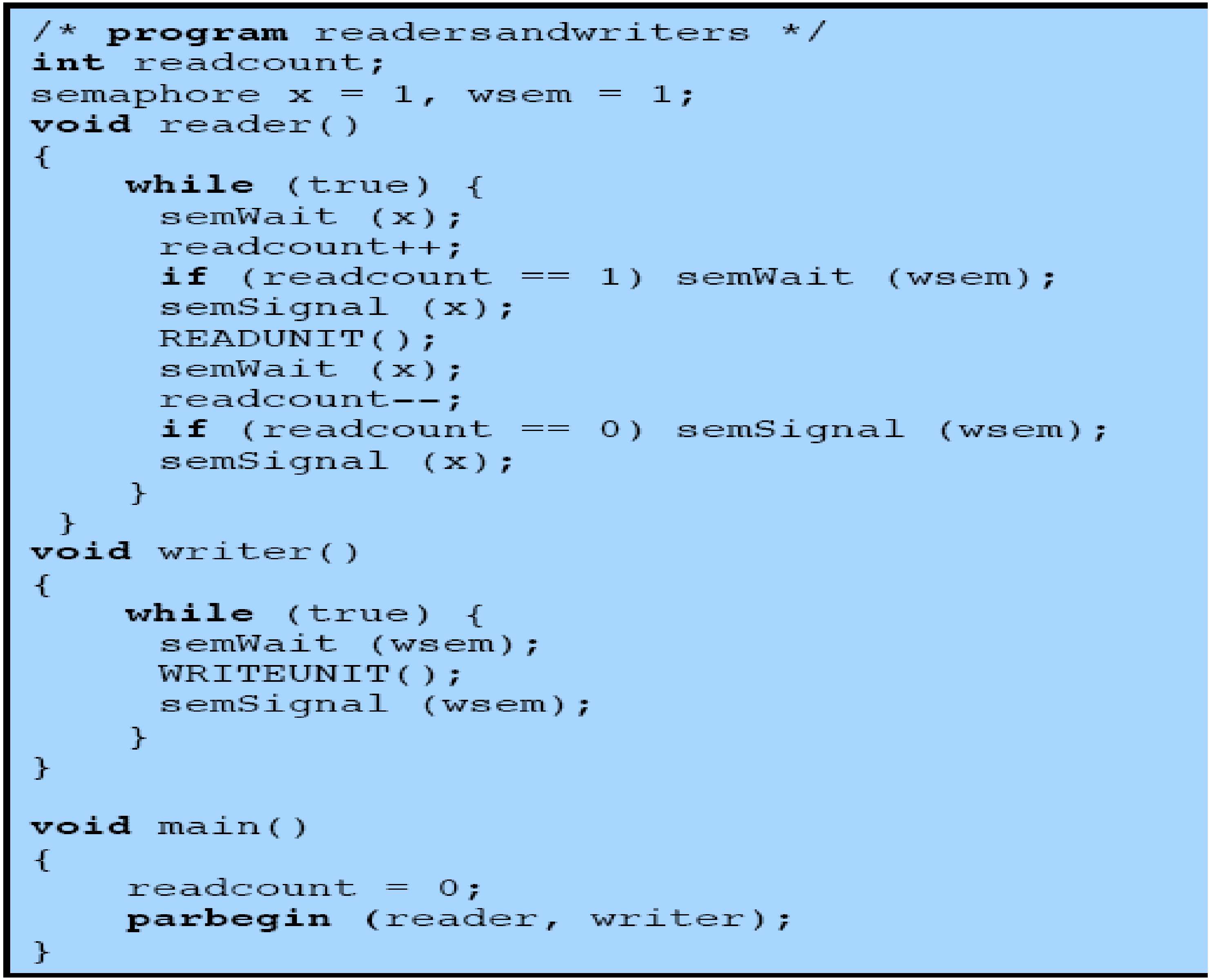
Initially filled with a number of null msg = equal to the capacity of the buffer

Shrink with each production
Grows with each consumption

Readers/Writers Problem

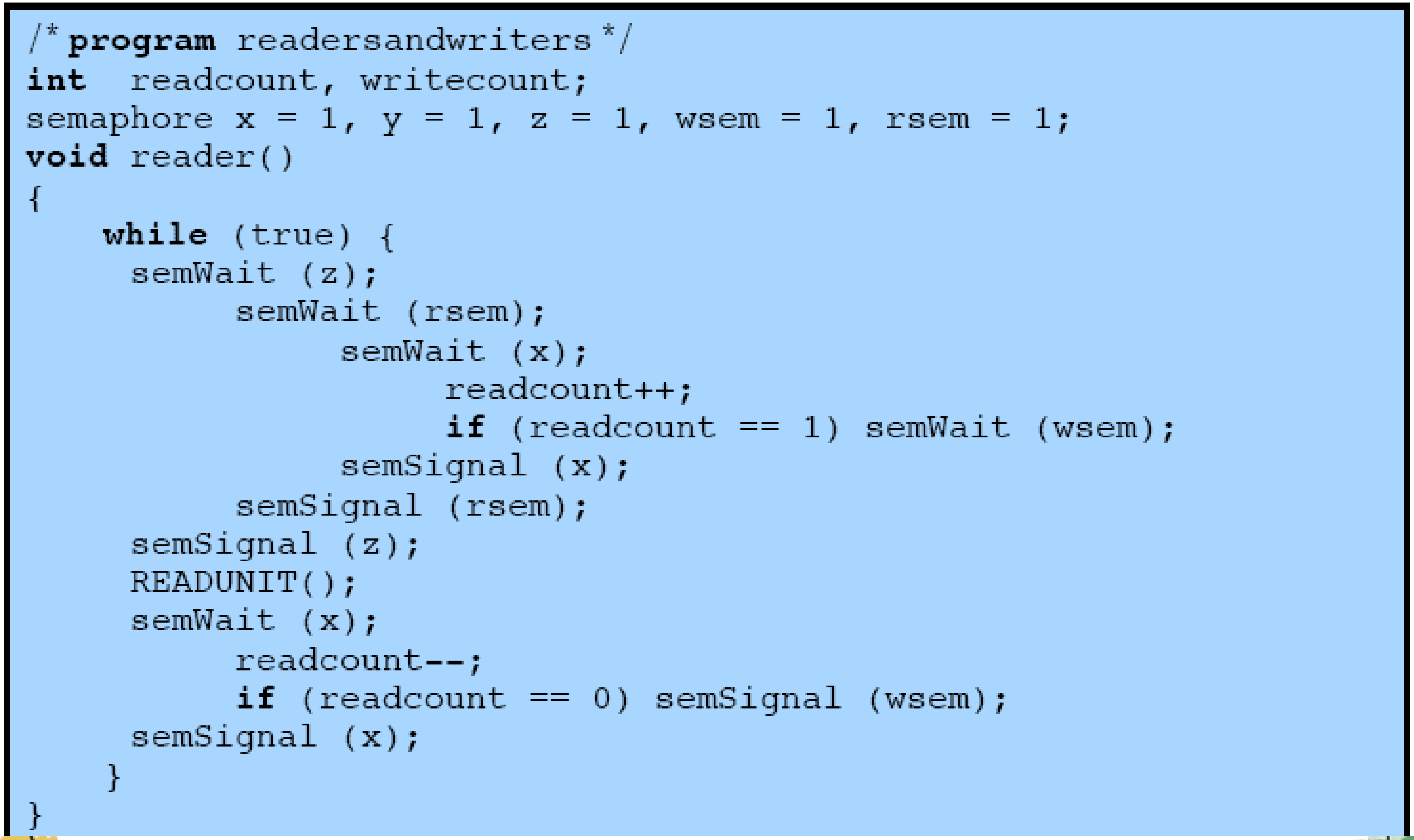
- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

Readers have Priority

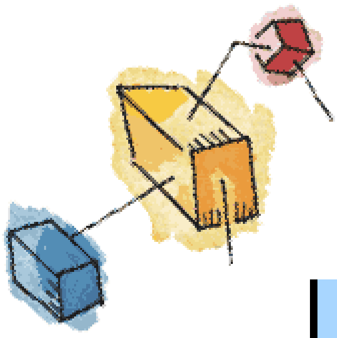


```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1) semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0) semSignal (wsem);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

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

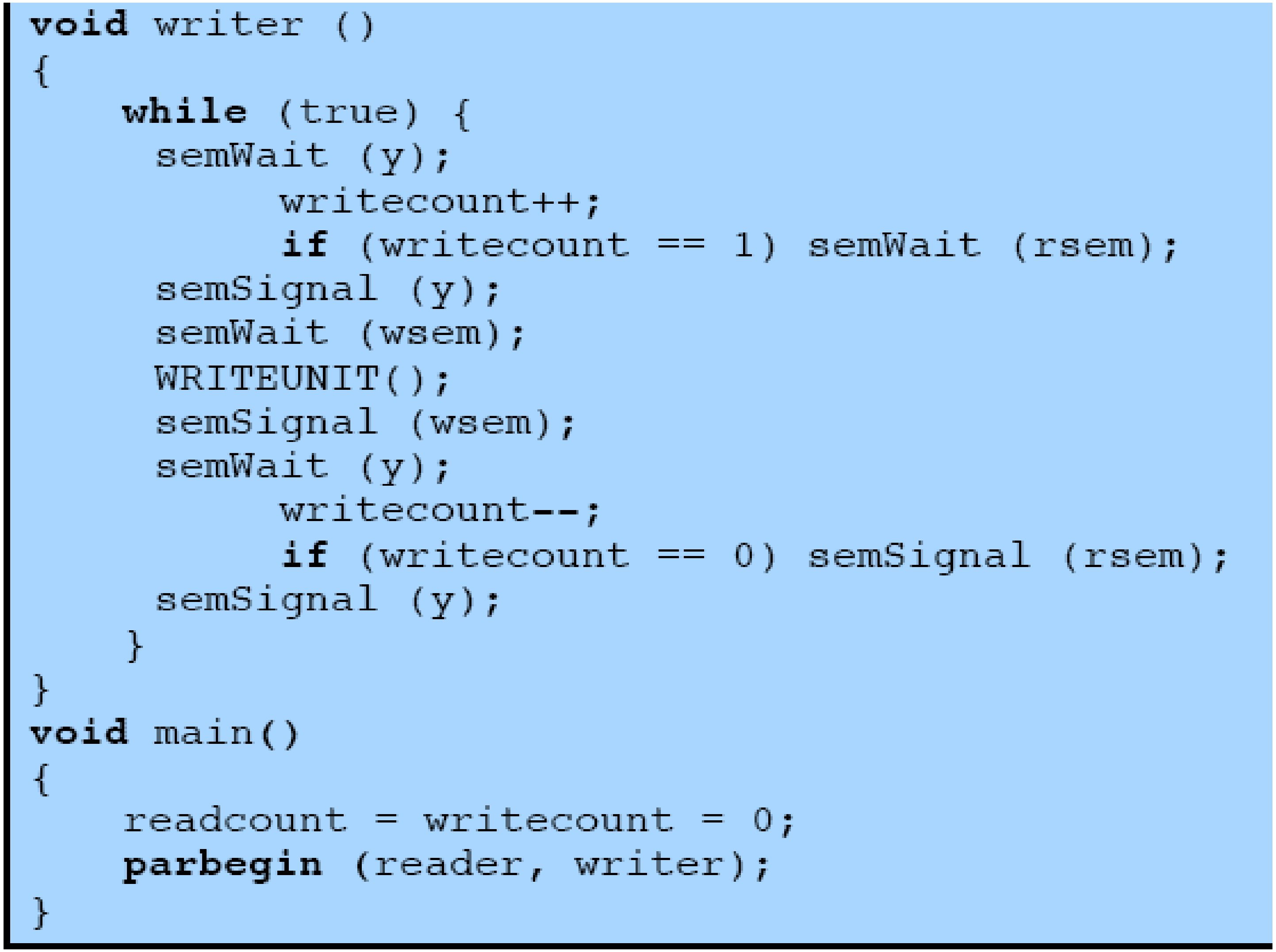
Writers have Priority



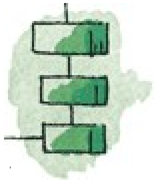
```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait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

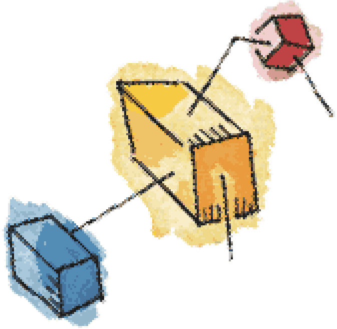


Writers have Priority

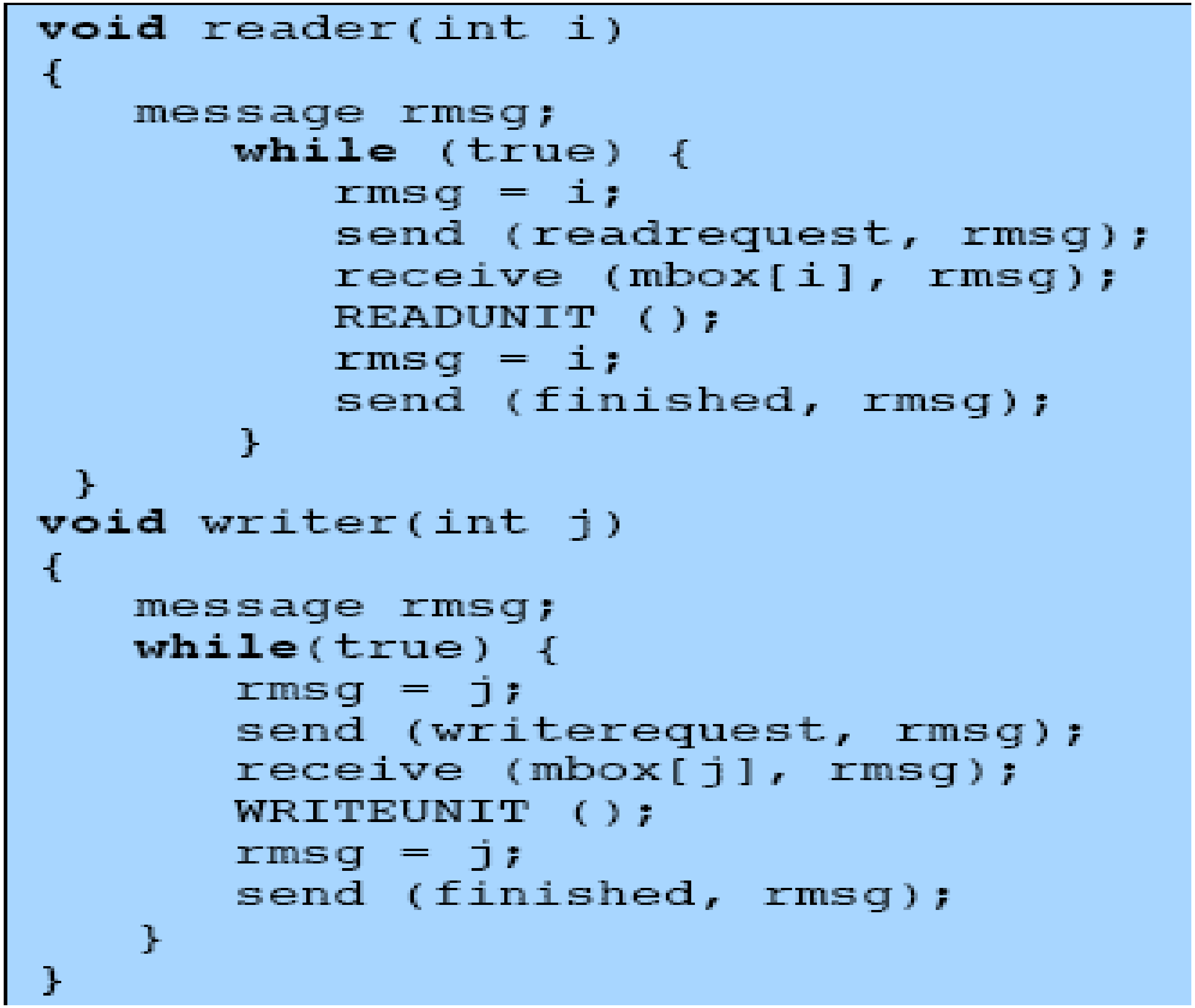


```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);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

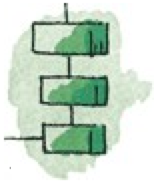




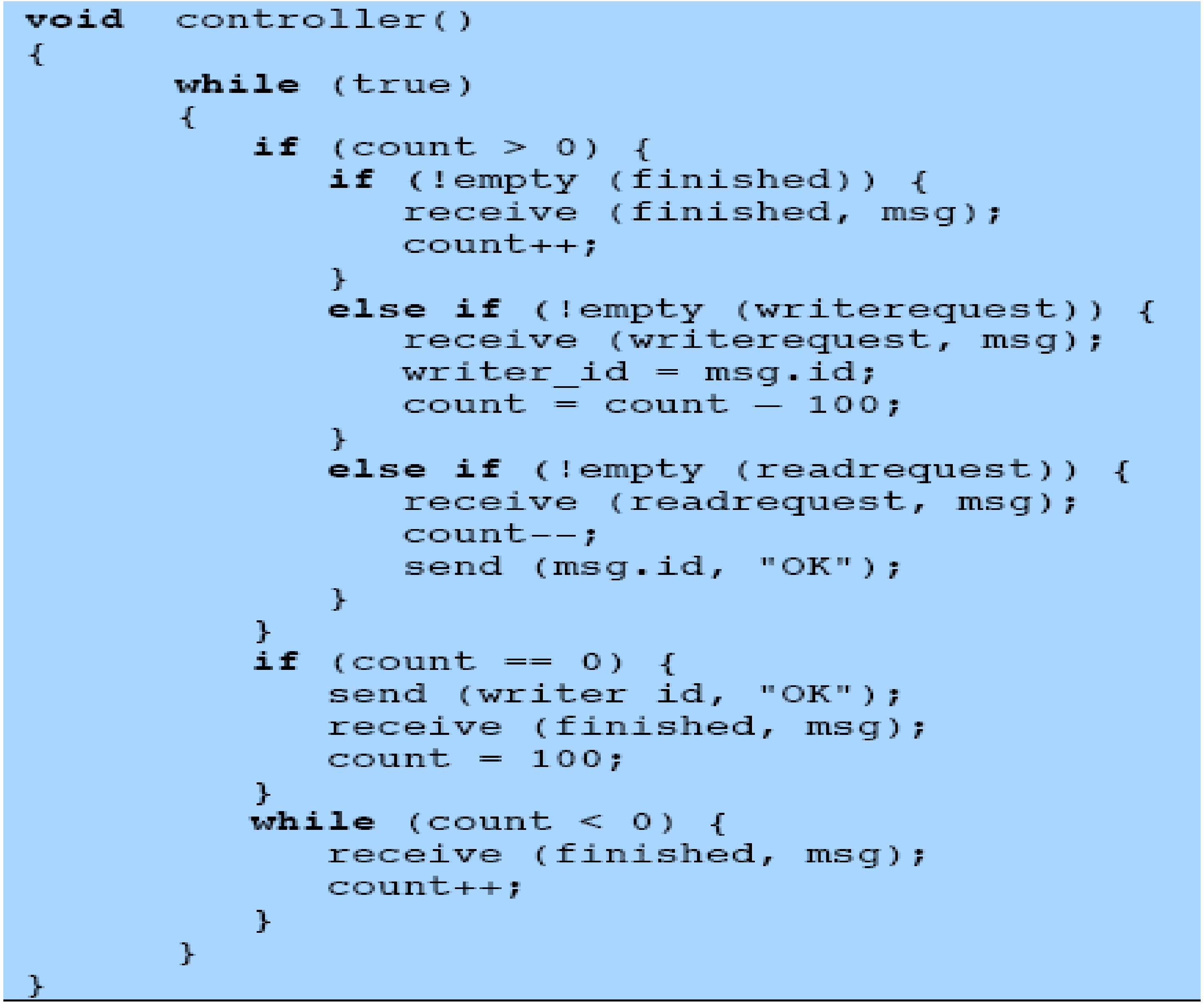
Message Passing



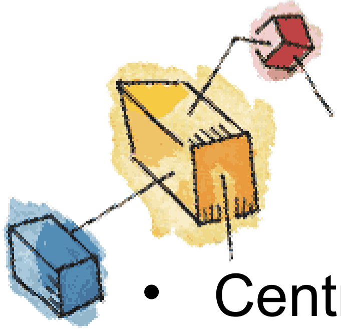
```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg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



Message Passing



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



Summary

- Central theme of modern OS: multiprogramming, multiprocessing, distributed processing.
  - Fundamental: concurrency
- Concurrent processes can communicate depends on whether they are indirect or directly aware of others' existence
- Mutual exclusion: a condition in which there are a set of concurrent processes, only one of which is able to access a given resources of perform a given function at any time.
  - Can be used to resolve competition of resources and synchronize processes
  - Machine instruction can be used to support mutual exclusion.
  - Also using semaphores and message facilities.