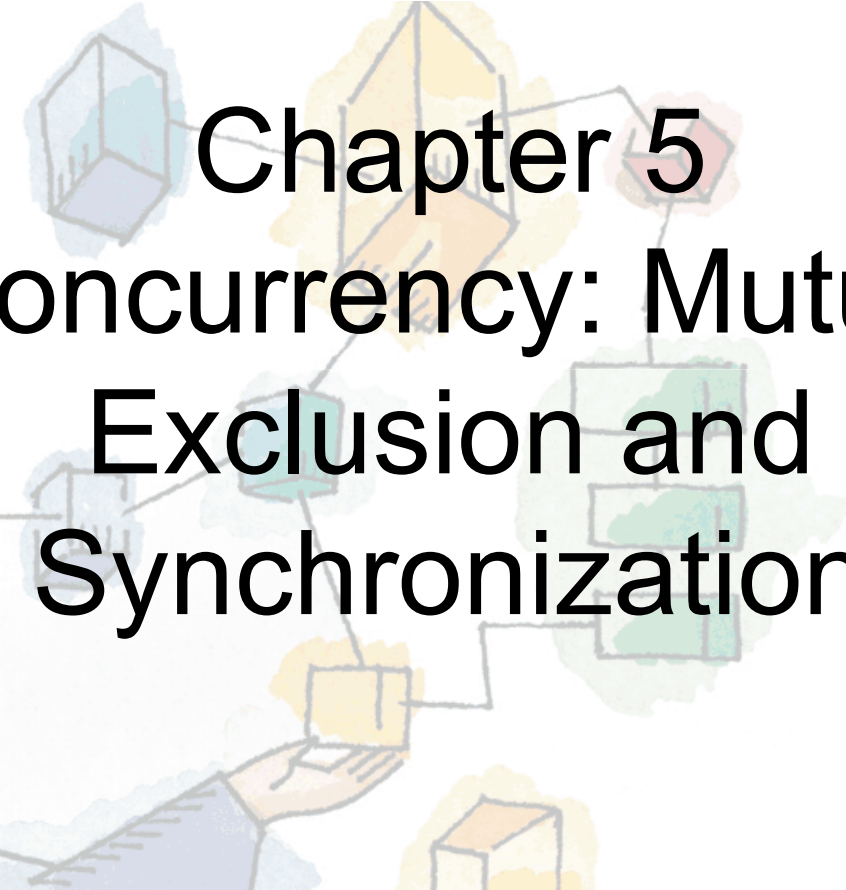
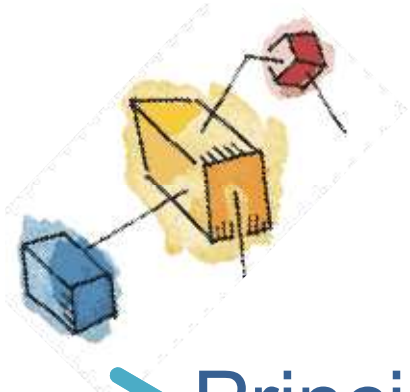


*Operating Systems:  
Internals and Design Principles, 6/E*  
William Stallings



**Chapter 5**  
**Concurrency: Mutual  
Exclusion and  
Synchronization**

Dave Bremer  
Otago Polytechnic, N.Z.  
©2008, Prentice Hall



# Roadmap

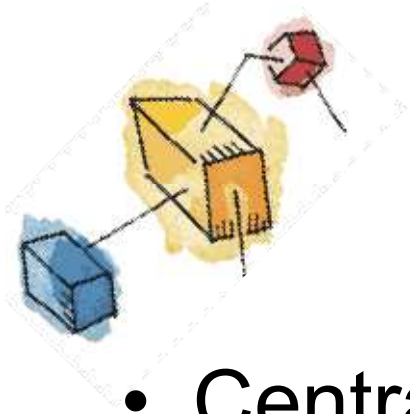
## → Principals of Concurrency

- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem





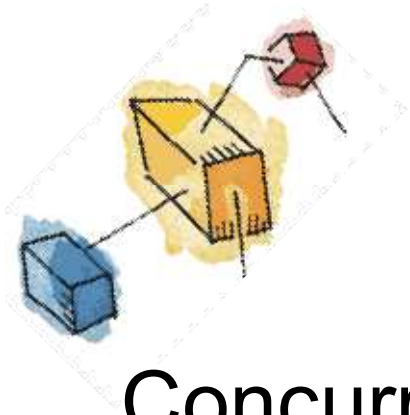
# Multiple Processes



- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes



# Concurrency



Concurrency arises in:

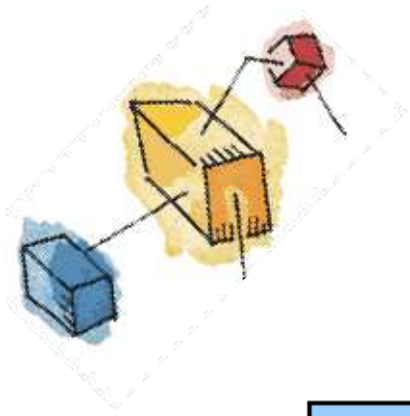
- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

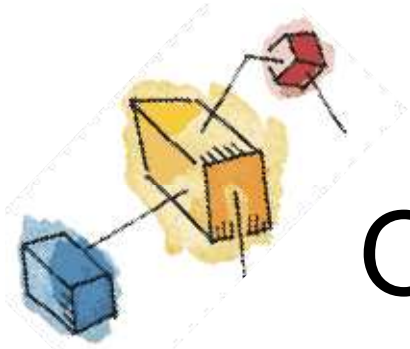


# Key Terms

Table 5.1 Some Key Terms Related to Concurrency

<b>atomic operation</b>	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
<b>critical section</b>	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.
<b>deadlock</b>	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
<b>livelock</b>	A situation in which two or more processes continuously change their states in response to changes in the other <u>process(es)</u> without doing any useful work.
<b>mutual exclusion</b>	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.
<b>race condition</b>	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.
<b>starvation</b>	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.





# Interleaving and Overlapping Processes

- Earlier (Ch2) we saw that processes may be interleaved on uniprocessors

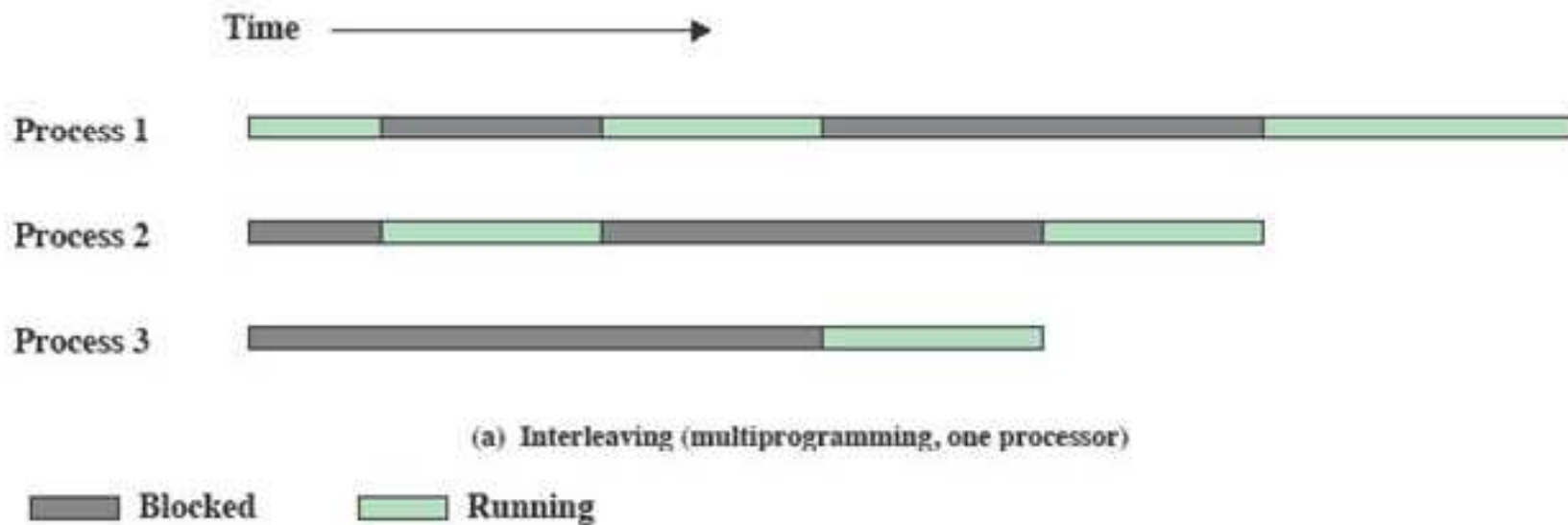
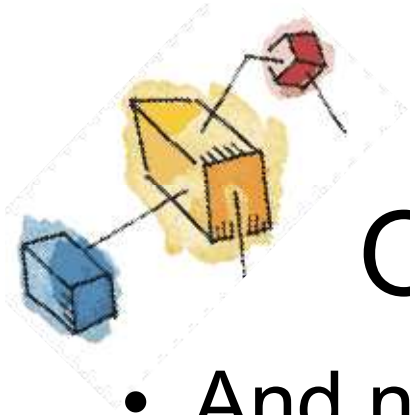


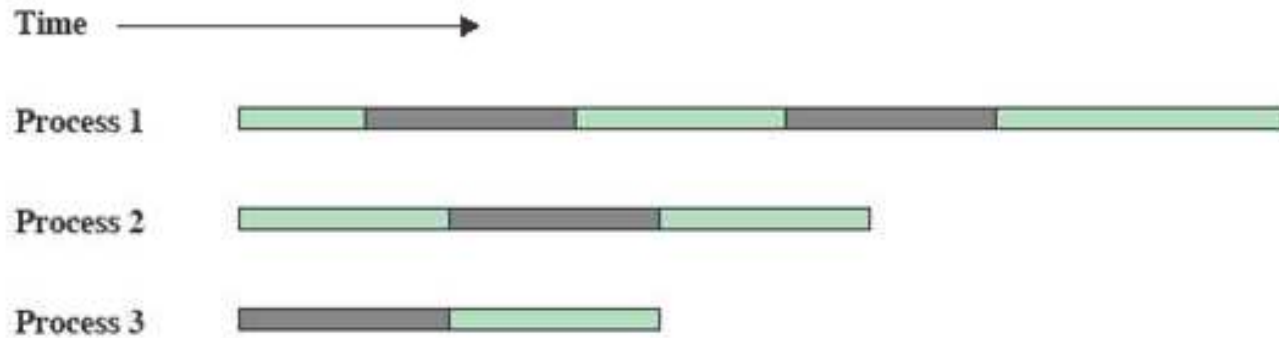
Figure 2.12 Multiprogramming and Multiprocessing





# Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors



(b) Interleaving and overlapping (multiprocessing; two processors)

Blocked Running

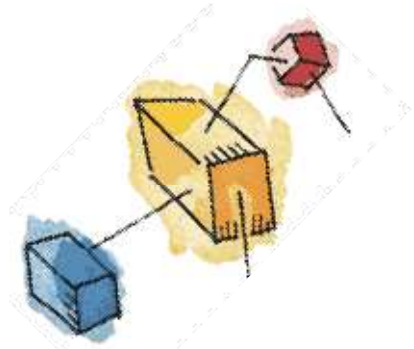
Figure 2.12 Multiprogramming and Multiprocessing



# Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors as results are not deterministic and reproducible.

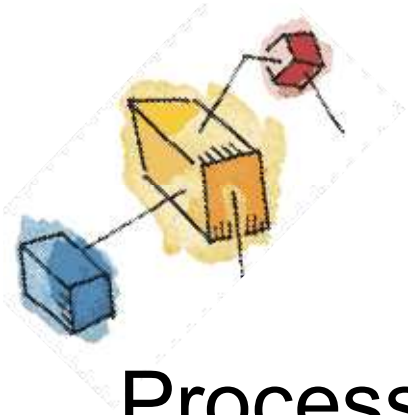




# A Simple Example

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





# A Simple Example: On a Multiprocessor

Process P1

Process P2

```

.
chin = getchar();
.
chout = chin;
putchar(chout);
.
.

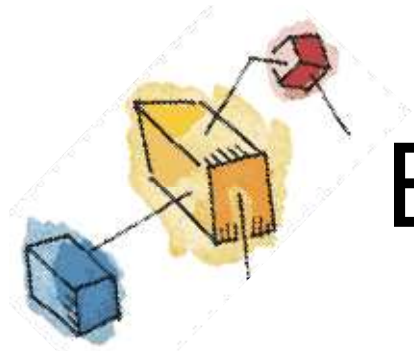
```

```

.
.
chin = getchar();
chout = chin;
.
putchar(chout);
.
.

```

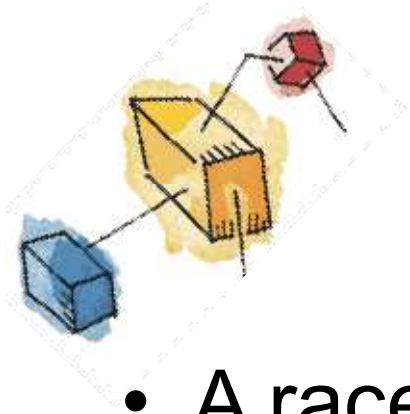




# Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

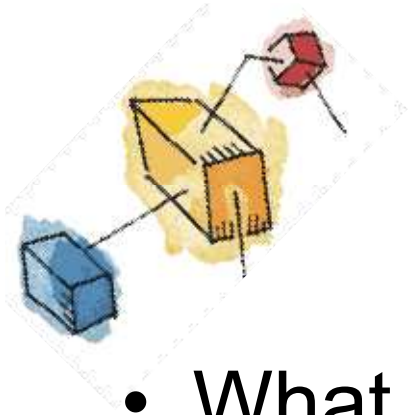




# Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.





# Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed



# 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	Competition	<ul style="list-style-type: none"><li>• Results of one process independent of the action of others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li></ul>
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li><li>• Data coherence</li></ul>
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Deadlock (consumable resource)</li><li>• Starvation</li></ul>



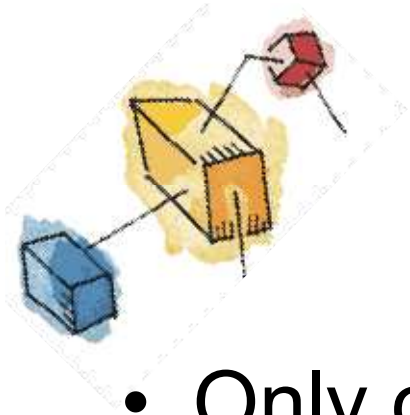


# Competition among Processes for Resources

Three main control problems:

- Need for Mutual Exclusion
  - Critical sections
- Deadlock
- Starvation

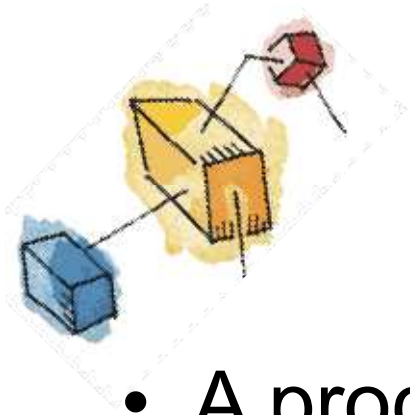




# Requirements for Mutual Exclusion

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

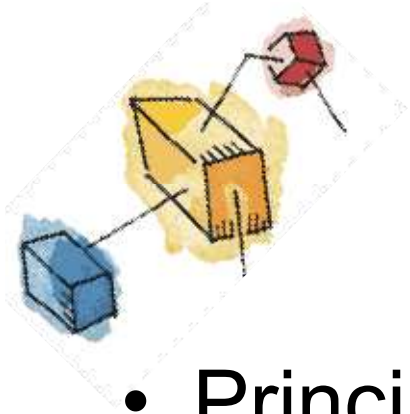




# Requirements for Mutual Exclusion

- 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



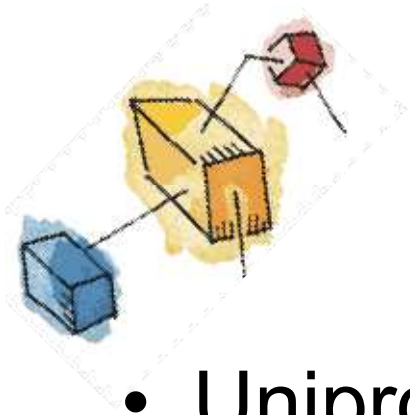


# Roadmap

- Principals of Concurrency
- ➔ Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem



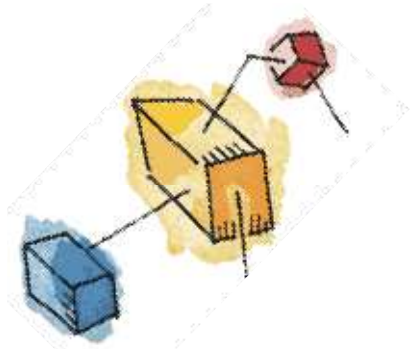
# Disabling Interrupts



- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture



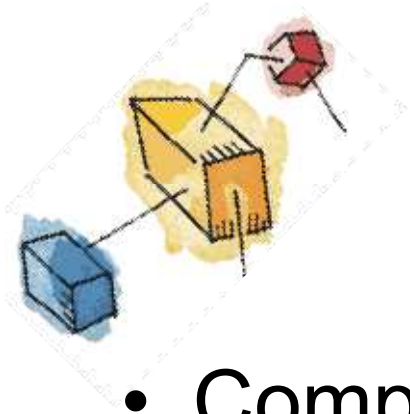
# Pseudo-Code



```
while (true) {  
    /* disable interrupts */;  
    /* critical section */;  
    /* enable interrupts */;  
    /* remainder */;  
}
```



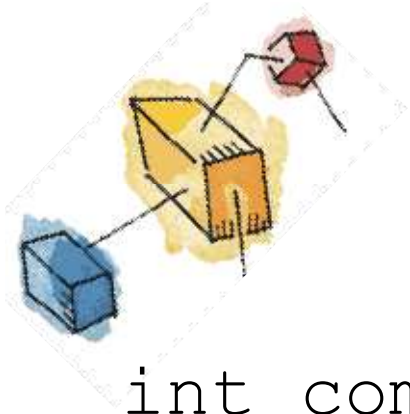
# Special Machine Instructions



- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
- Exchange Instruction

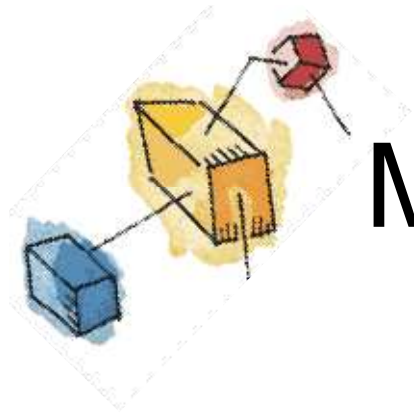


# Compare&Swap Instruction



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





# Mutual Exclusion (fig 5.2)

```
/* 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



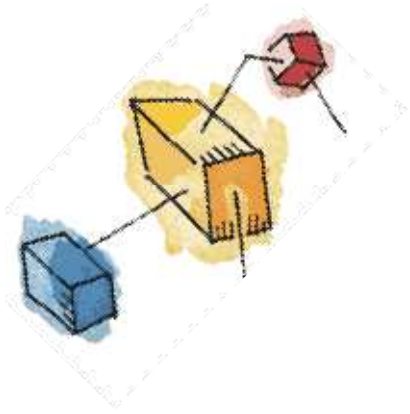


# Exchange instruction

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



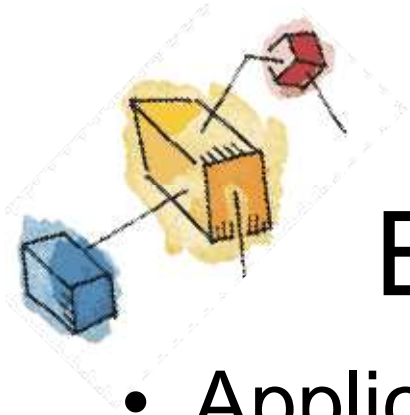
# Exchange Instruction (fig 5.2)



```
/* 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





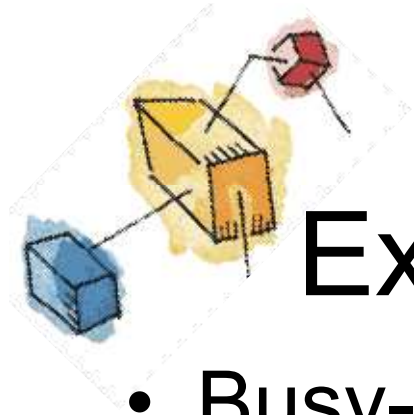
# Hardware Mutual Exclusion: Advantages

- 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



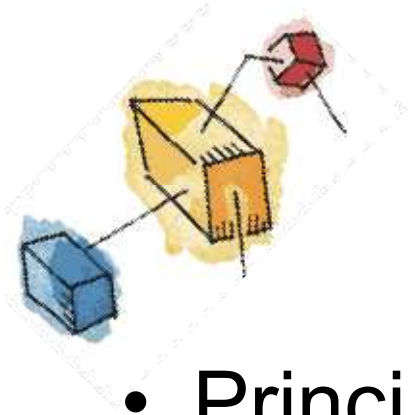


# Hardware Mutual Exclusion: Disadvantages



- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Some process could indefinitely be denied access.
- Deadlock is possible





# Roadmap

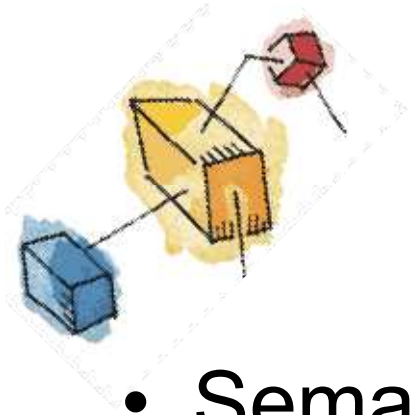
- Principals of Concurrency
- Mutual Exclusion: Hardware Support

## → Semaphores

- Monitors
- Message Passing
- Readers/Writers Problem



# Semaphore



- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (`semWait`)
  - increment. (`semSignal`)



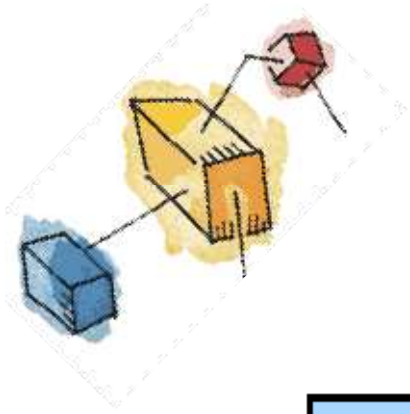
# Semaphore Primitives

```
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 */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives



# Binary Semaphore Primitives

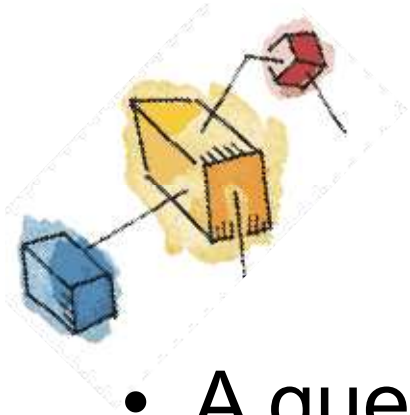


```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s)
{
    if (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

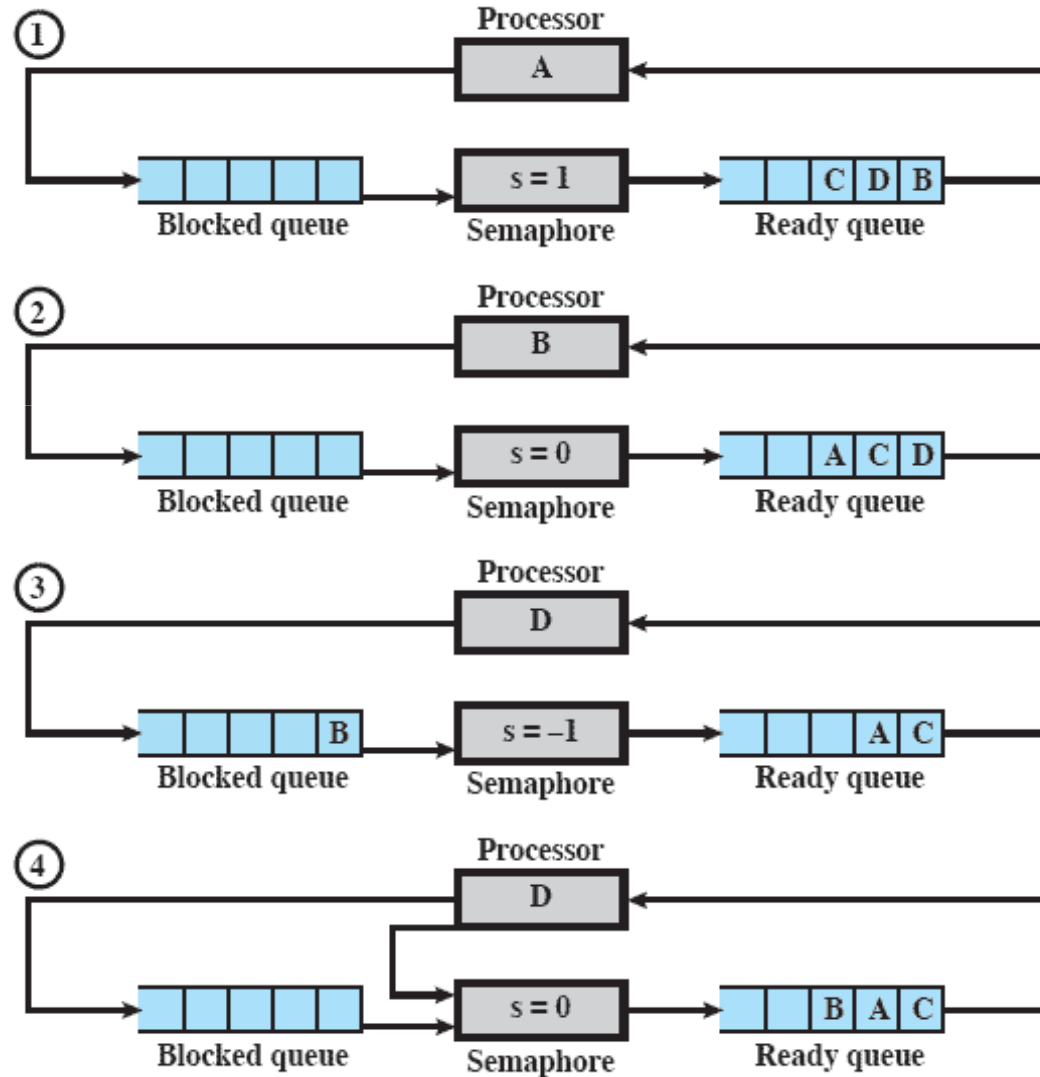
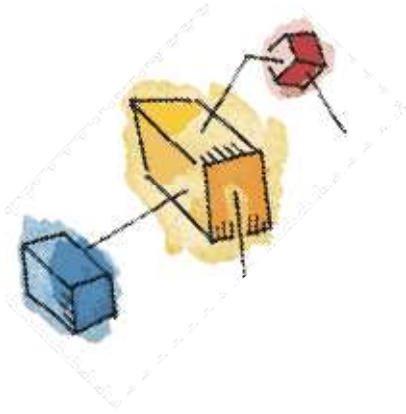
# Strong/Weak Semaphore

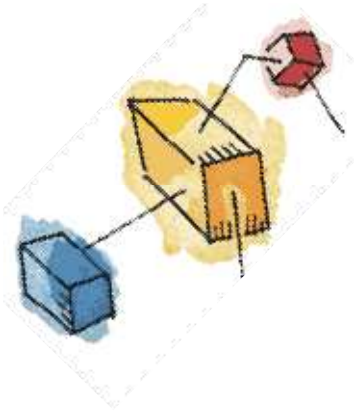


- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- **Strong Semaphores** use FIFO
- **Weak Semaphores** don't specify the order of removal from the queue



# Example of Strong Semaphore Mechanism





# Example of Semaphore Mechanism

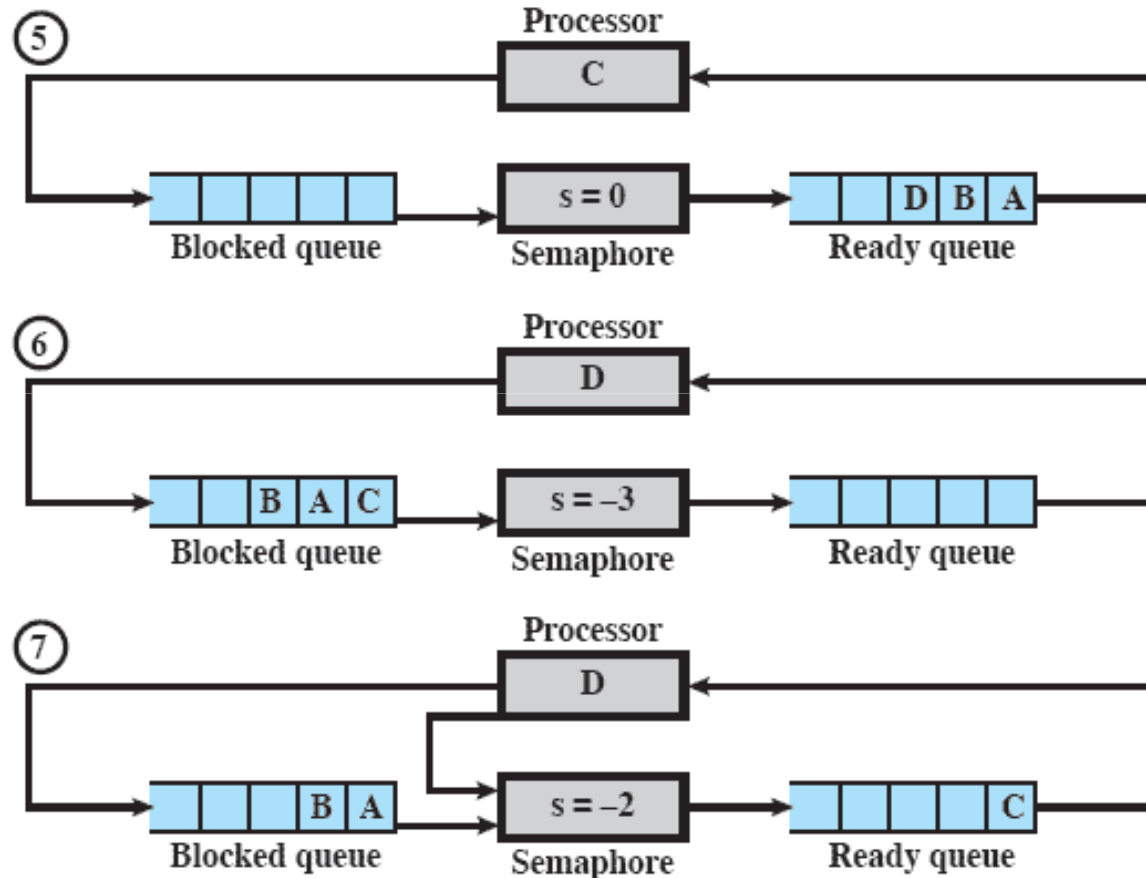
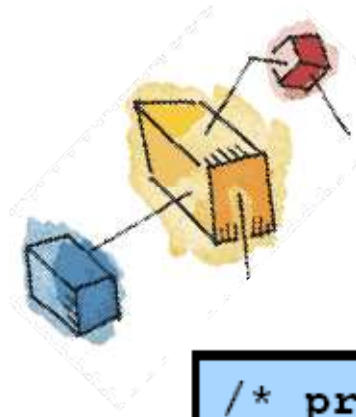


Figure 5.5 Example of Semaphore Mechanism





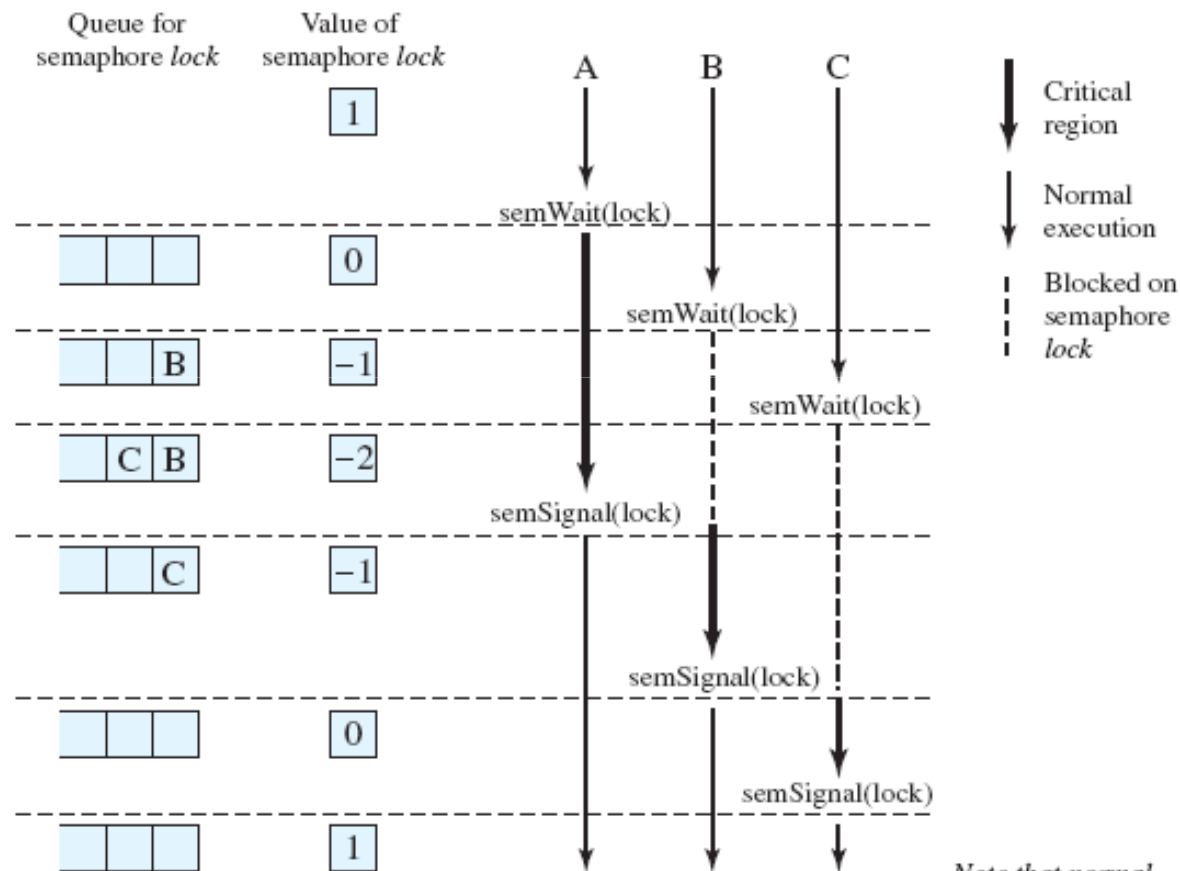
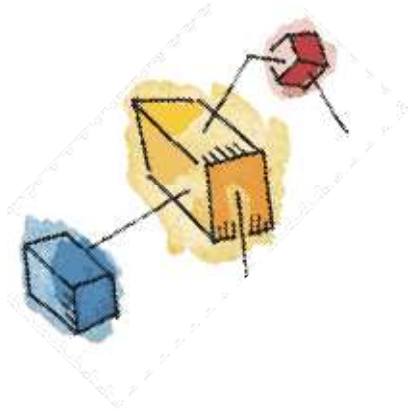
# Mutual Exclusion Using Semaphores

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

Figure 5.6 Mutual Exclusion Using Semaphores



# Processes Using Semaphore

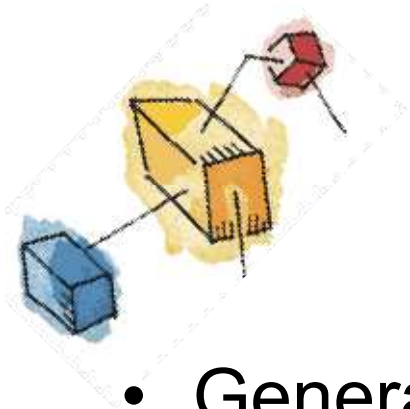


*Note that normal execution can proceed in parallel but that critical regions are serialized.*



**Figure 5.7** Processes Accessing Shared Data Protected by a Semaphore





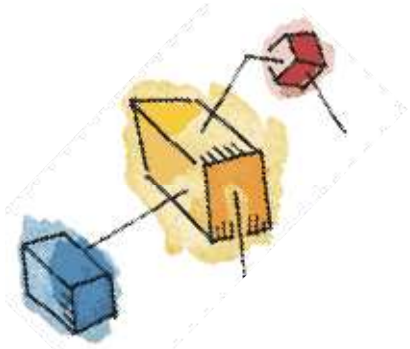
# Producer/Consumer Problem

- General Situation:
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time
- The Problem:
  - Ensure that the Producer can't add data into full buffer and consumer can't remove data from empty buffer



Producer/Consumer Animation





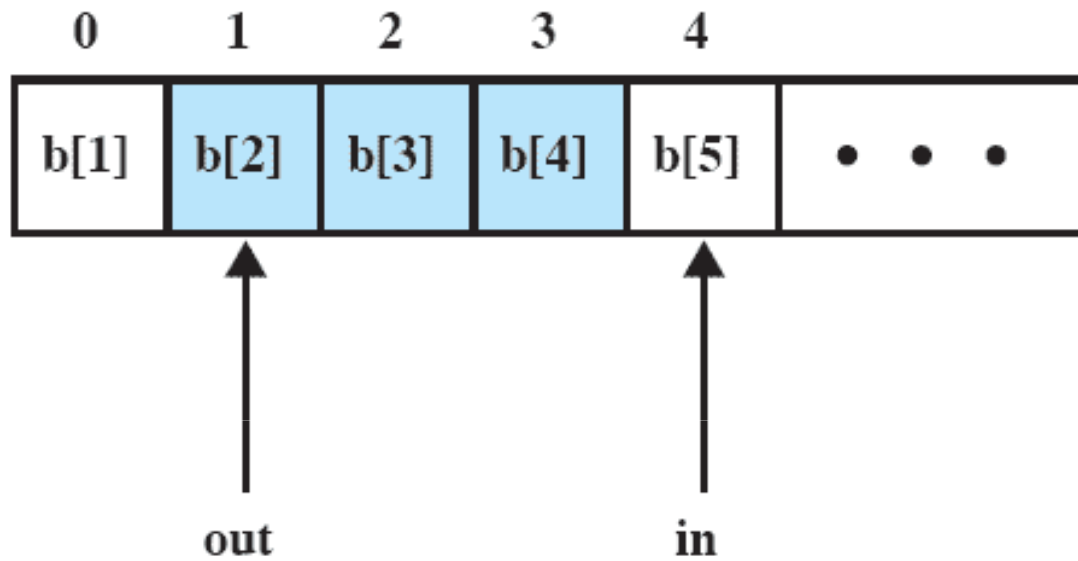
# Functions

- Assume an infinite buffer **b** with a linear array of elements

Producer	Consumer
<pre>while (true) {     /* produce item v */     b[in] = v;     in++; }</pre>	<pre>while (true) {     while (in &lt;= out)         /*do nothing */;     w = b[out];     out++;     /* consume item w */ }</pre>



# Buffer



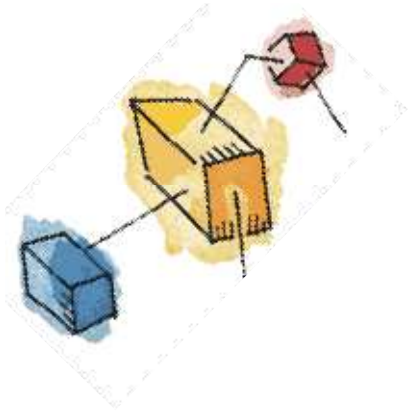
Note: shaded area indicates portion of buffer that is occupied

**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);
}
```

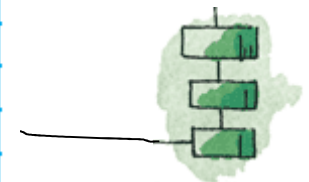
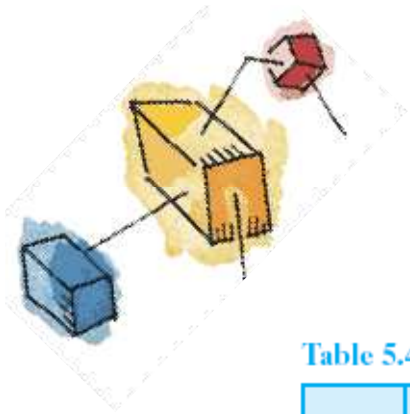


# Possible Scenario

**Table 5.4** Possible Scenario for the Program of Figure 5.9

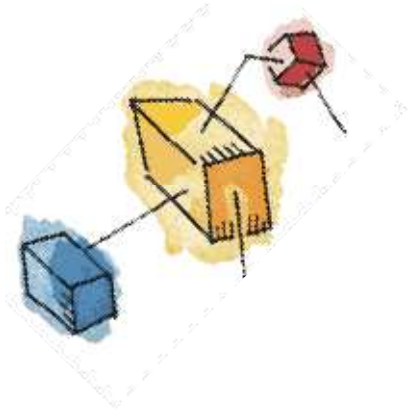
	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		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

*NOTE:* White areas represent the critical section controlled by semaphore s

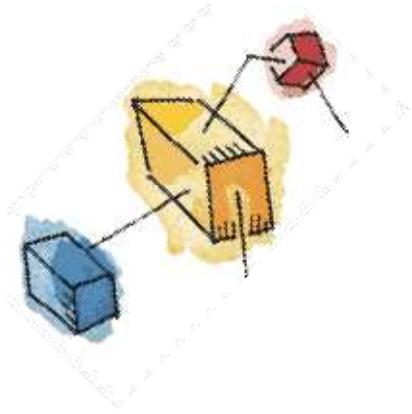


# Correct Solution

```
/* 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

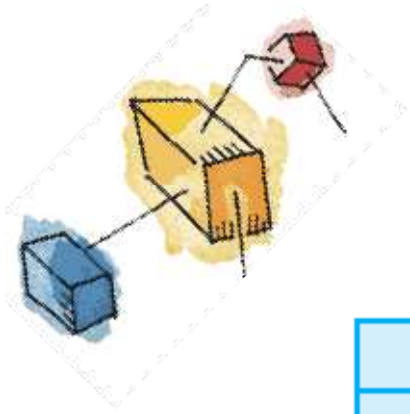


```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
  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



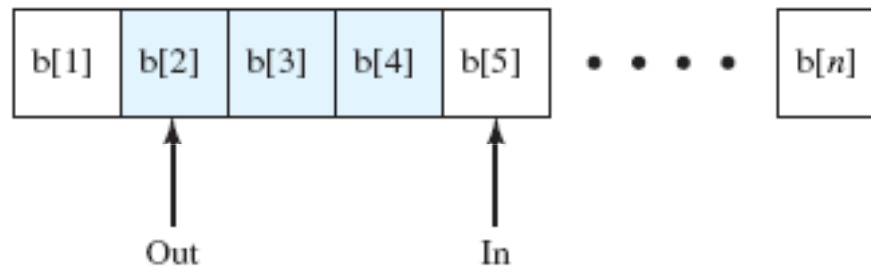
Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores



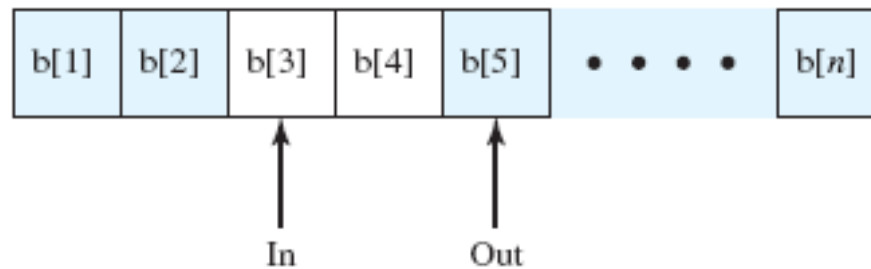


# Bounded Buffer

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



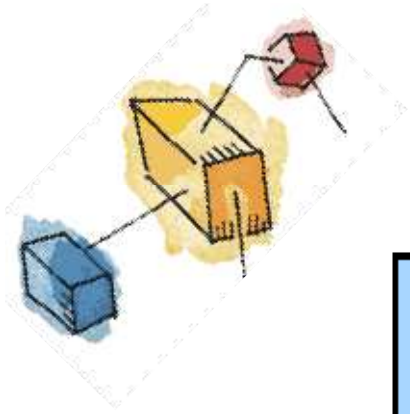
(a)



(b)



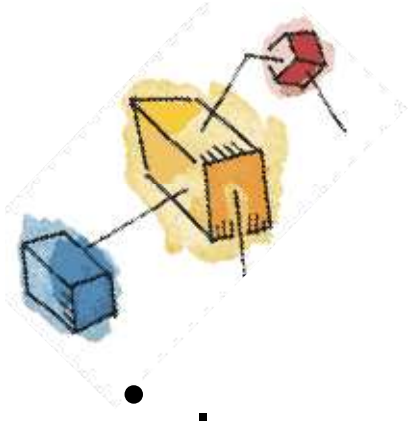
**Figure 5.12** Finite Circular Buffer for the Producer/Consumer Problem



# Semaphores

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```



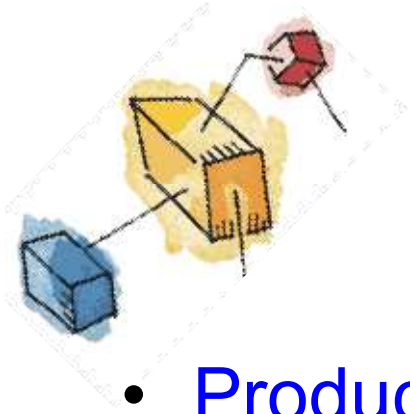


# Functions in a Bounded Buffer

Producer	Consumer
<pre>while (true) {     /* produce item v */     while ((in + 1) % n == out) /*         do nothing */;     b[in] = v;     in = (in + 1) % n }</pre>	<pre>while (true) {     while (in == out)         /* do nothing */;     w = b[out];     out = (out + 1) % n;     /* consume item w */ }</pre>

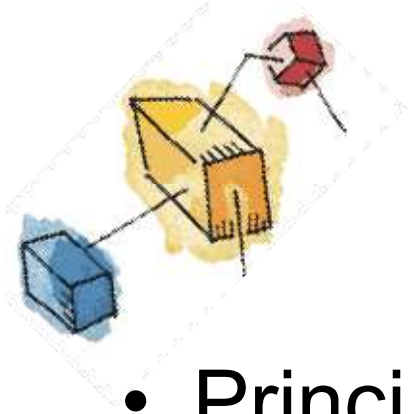


# Demonstration Animations



- [Producer/Consumer](#)
  - Illustrates the operation of a producer-consumer buffer.
- [Bounded-Buffer Problem Using Semaphores](#)
  - Demonstrates the bounded-buffer consumer/producer problem using semaphores.





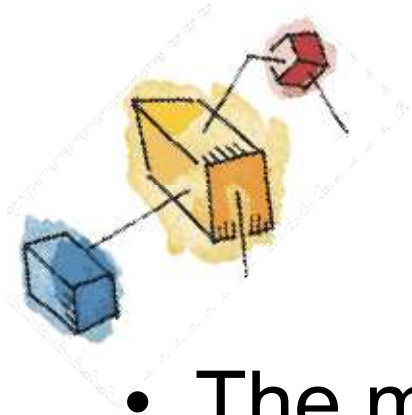
# Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores

## → Monitors

- Message Passing
- Readers/Writers Problem





# Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.



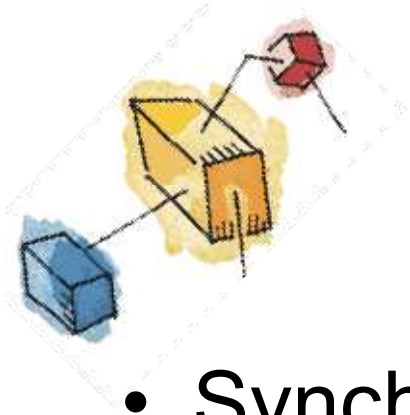
# Chief characteristics



- 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



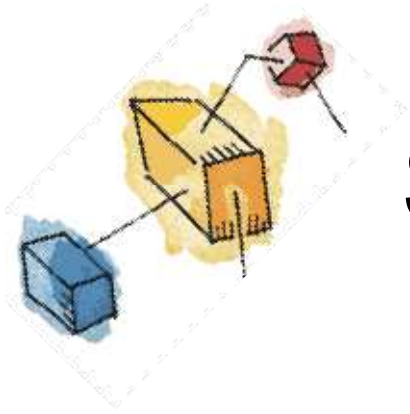
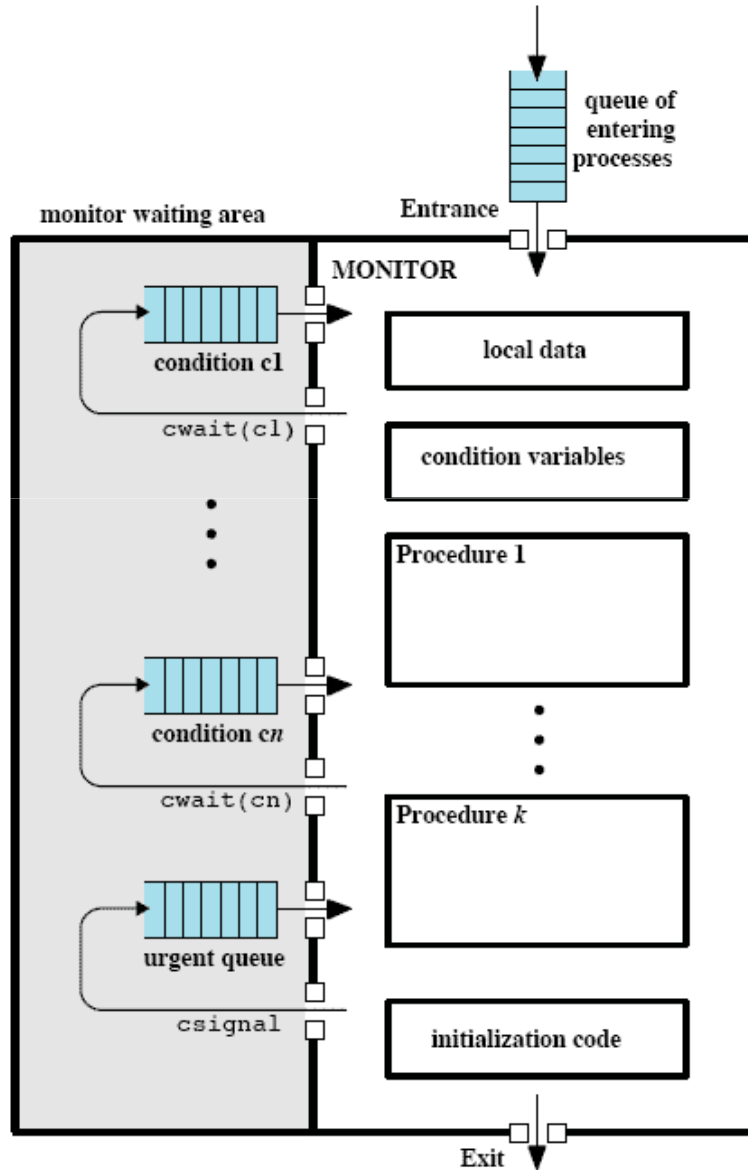
# Synchronization



- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - $Cwait(c)$ : 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





# Bounded Buffer Solution Using Monitor

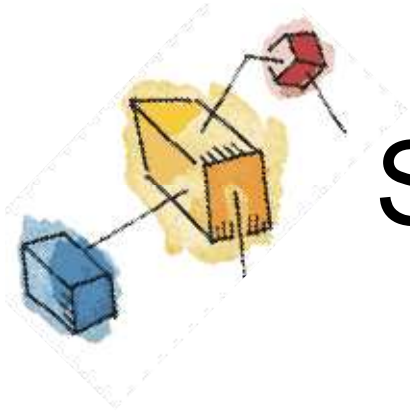
```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                /* space for N items */
int nextin, nextout;           /* buffer pointers */
int count;                      /* number of items in buffer */
cond notfull, notempty;       /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);           /* resume any waiting consumer */
}

void take (char x)
{
    if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);           /* resume any waiting producer */
}

/* monitor body */
nextin = 0; nextout = 0; count = 0;
/* buffer initially empty */
}
```

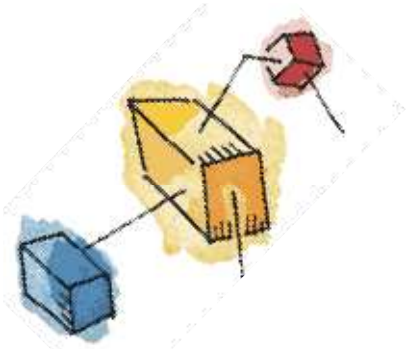
# Solution Using Monitor



```
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);
}
```



# Bounded Buffer Monitor

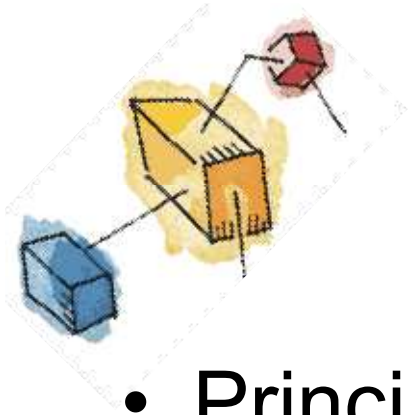


```
void append (char x)
{
    while(count == N) cwait(notfull);    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                               /* one more item in buffer */
    cnotify(notempty);                    /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                               /* one fewer item in buffer */
    cnotify(notfull);                     /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor



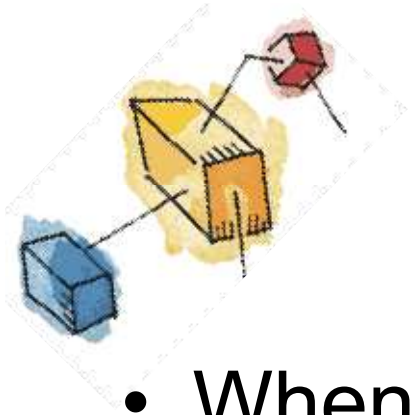


# Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem



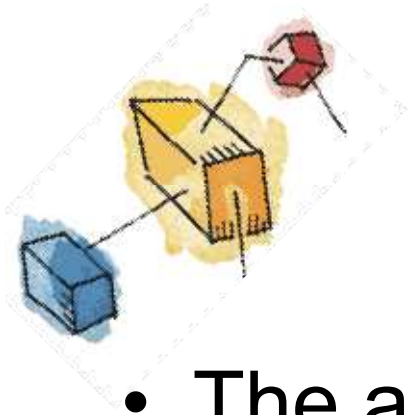
# Process Interaction



- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory *and* with distributed systems



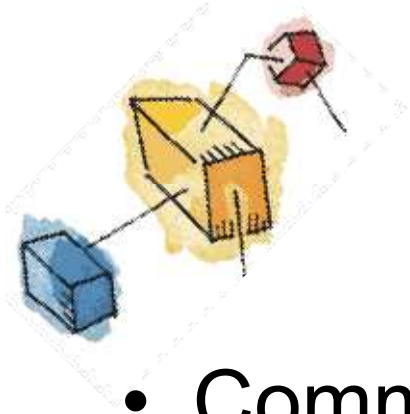
# Message Passing



- The actual function of message passing is normally provided in the form of a pair of primitives:
  - send (destination, message)
  - receive (source, message)

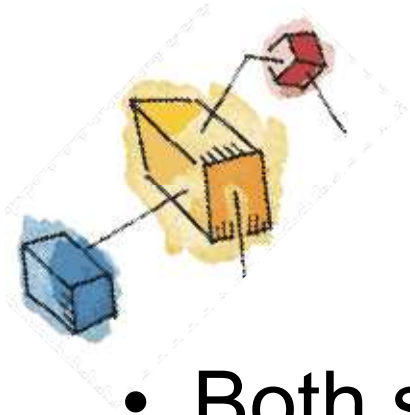


# Synchronization



- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - Sender and receiver may or may not be blocking (waiting for message)

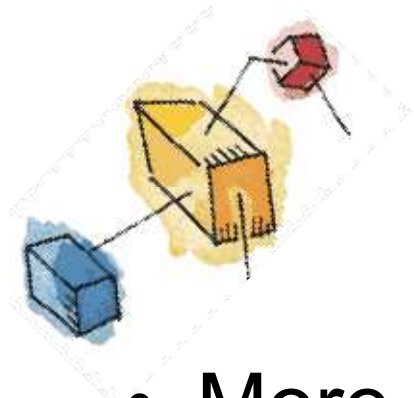




# Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Known as a *rendezvous*
- Allows for tight synchronization between processes.



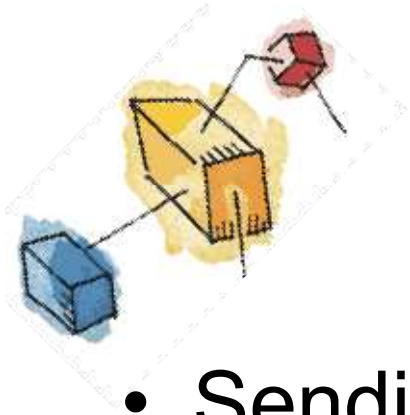


# Non-blocking Send

- More natural for many concurrent programming tasks.
- 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

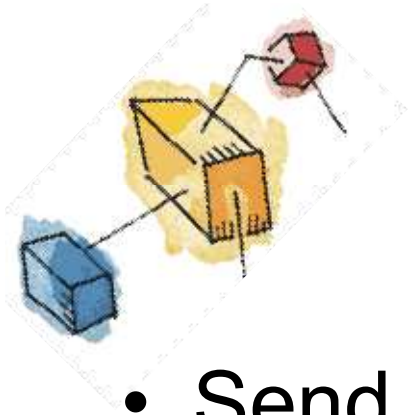


# Addressing



- Sendin process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing



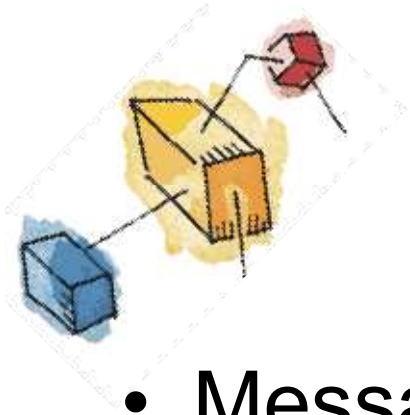


# Direct Addressing

- 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

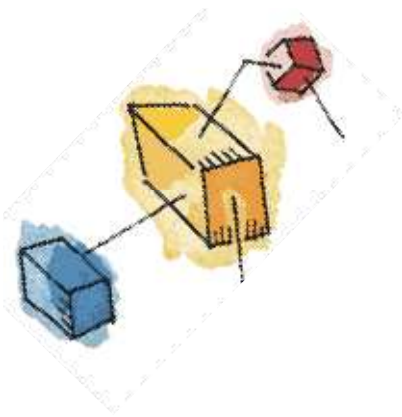


# Indirect addressing



- 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

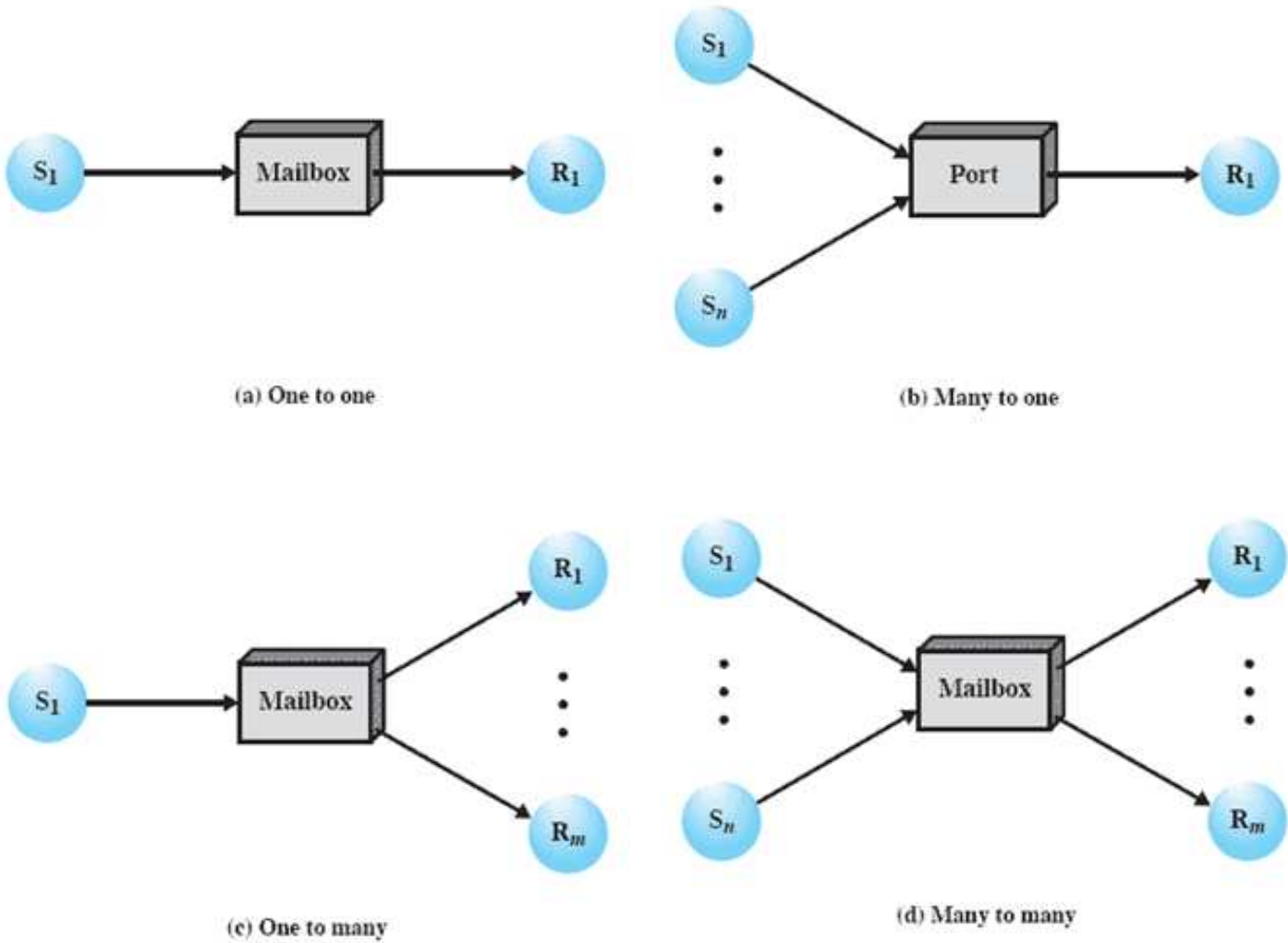


Figure 5.18 Indirect Process Communication

# General Message Format

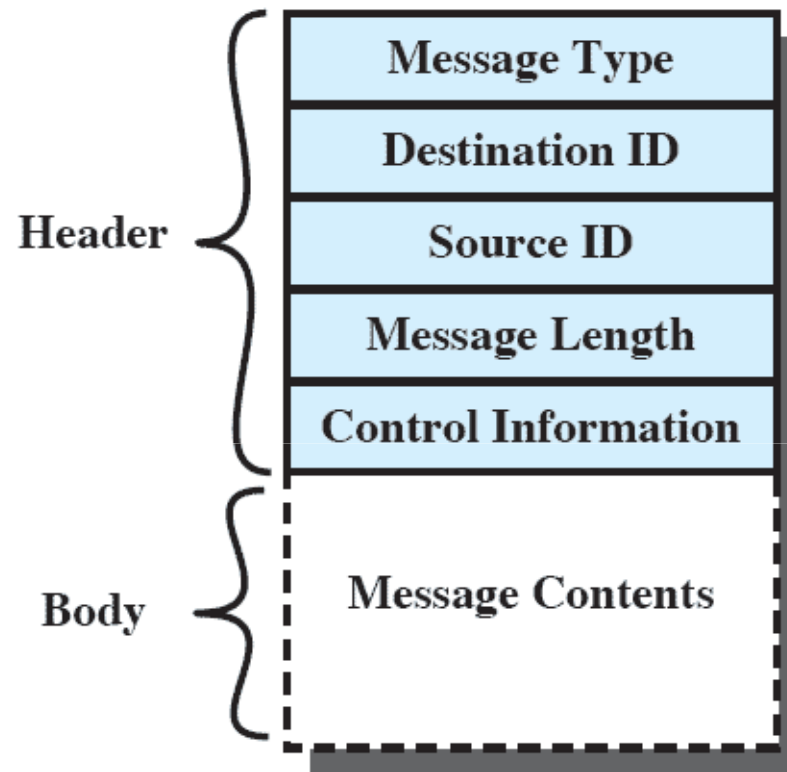
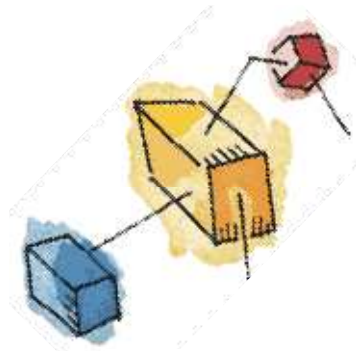


Figure 5.19 General Message Format



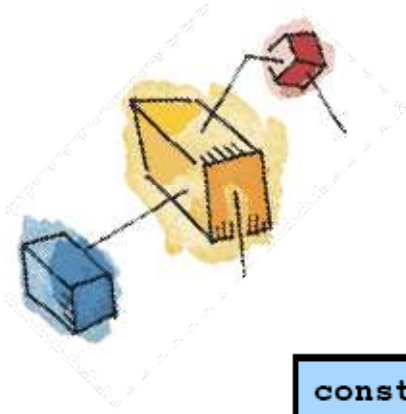
# Mutual Exclusion Using Messages

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

Figure 5.20 Mutual Exclusion Using Messages



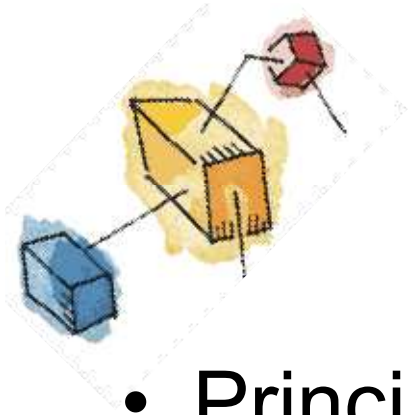
# Producer/Consumer Messages



```
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);
}
```





# Roadmap

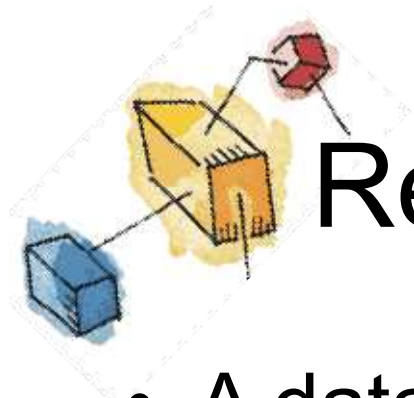
- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing

→ Readers/Writers Problem





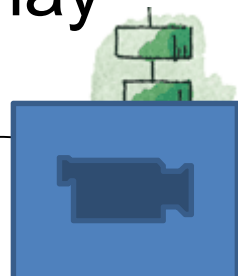
# Readers/Writers Problem



- A data area is shared among many processes
  - Some processes only read the data area, some only write to the area
- Conditions to satisfy:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.

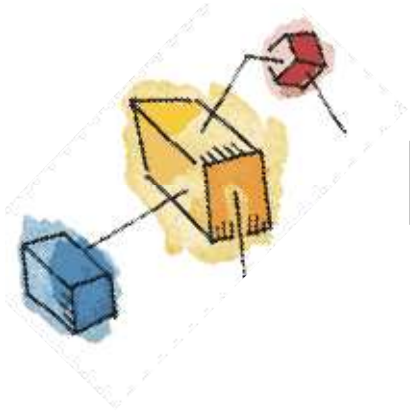


interaction of readers and writers.



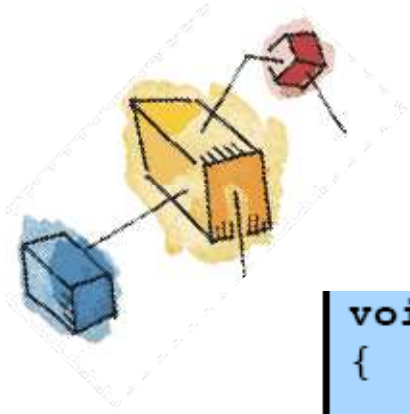
# Readers have Priority

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



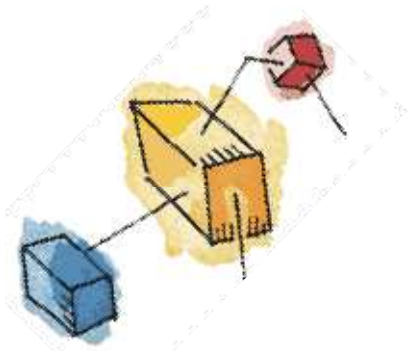
# Writers have Priority

```
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);
}
```

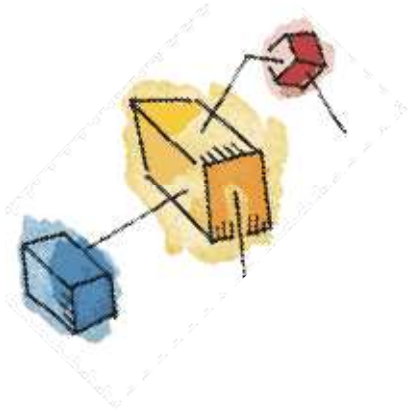


# Message Passing



```
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);
    }
}
```

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



# 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++;
    }
  }
}
```

