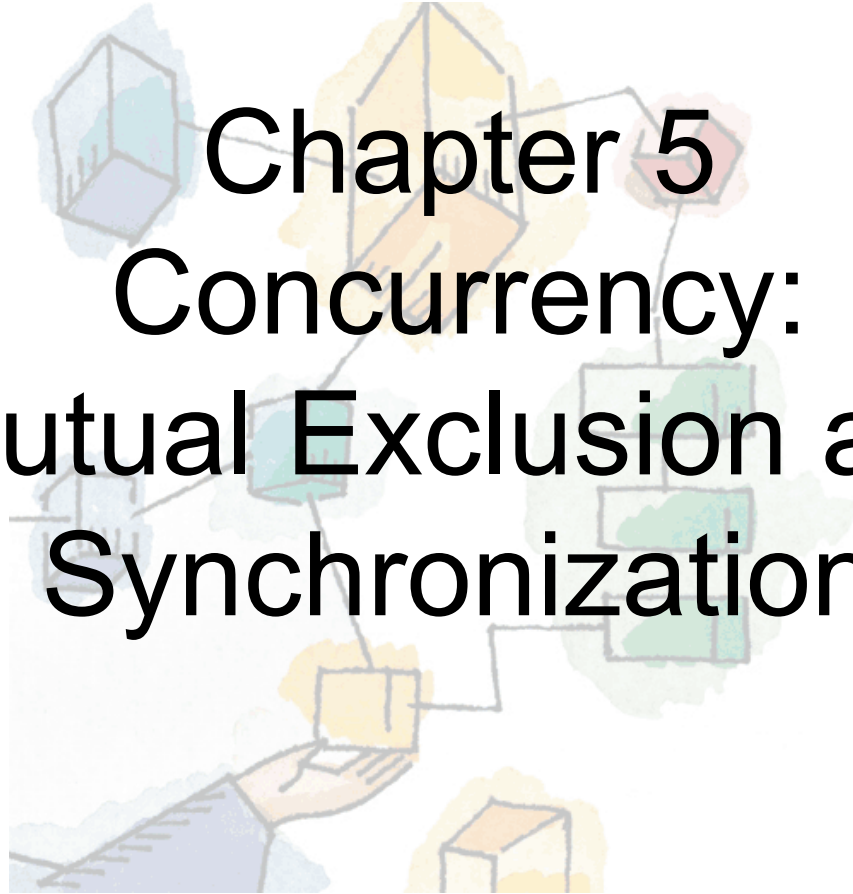
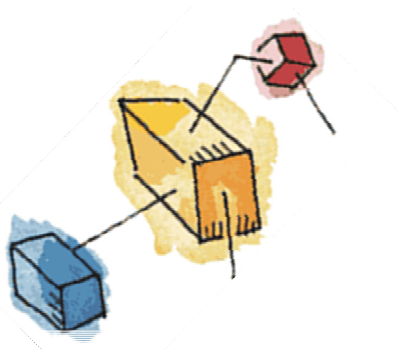


*Operating Systems:
Internals and Design Principles*
William Stallings

Chapter 5
Concurrency:
Mutual Exclusion and
Synchronization



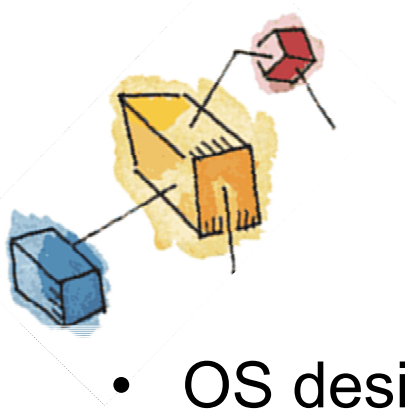


Roadmap

→ Principles of Concurrency

- Mutual Exclusion
- Semaphores
- IPC - Message Passing
- Readers/Writers Problem





Multiple Processes

- OS design is concerned with the management of processes and threads in different systems.
 - Multiprogramming
 - Multiple processes, one core
 - Multiprocessing
 - Multiple processes, multiple cores (shared memory)
 - Distributed Processing
 - Multiple processes, multiple nodes (distributed memory)





Interleaving and Overlapping Processes

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

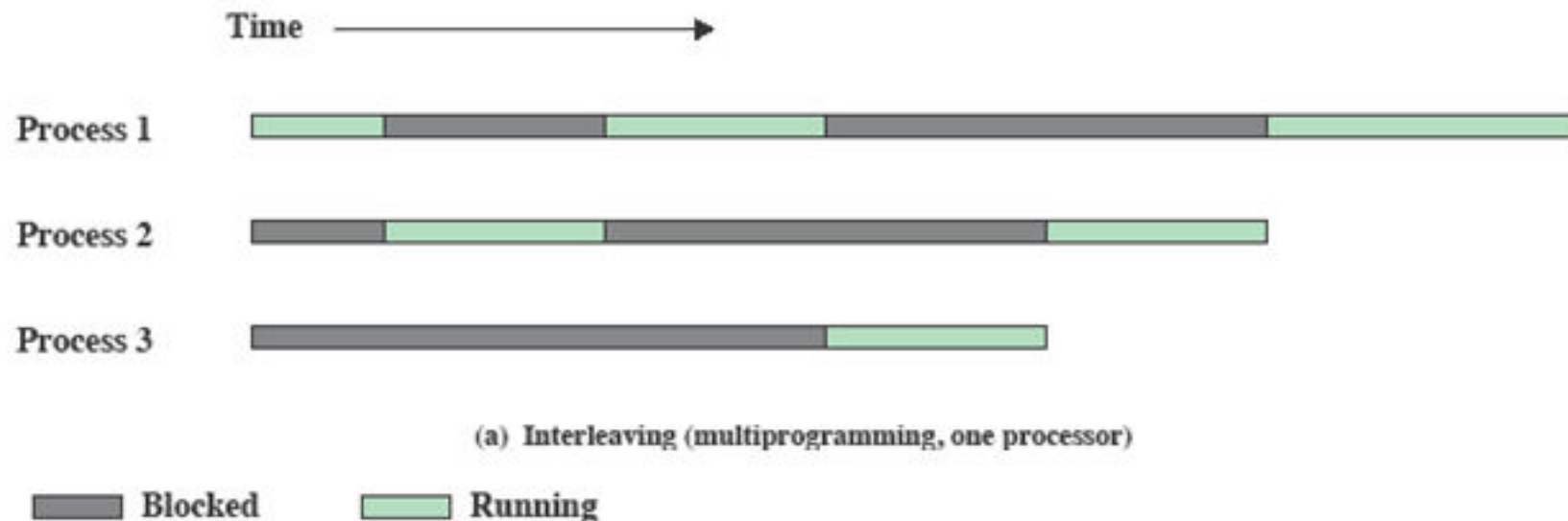
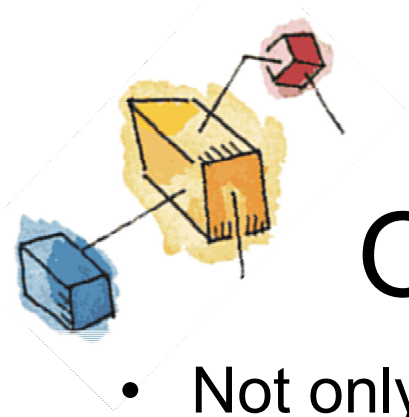


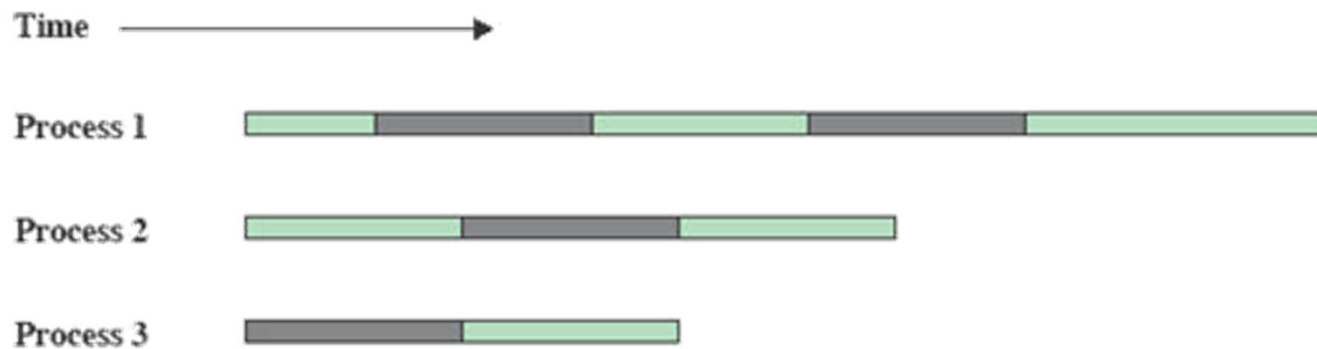
Figure 2.12 Multiprogramming and Multiprocessing





Interleaving and Overlapping Processes

- Not only interleaved but overlapped on multi-processors

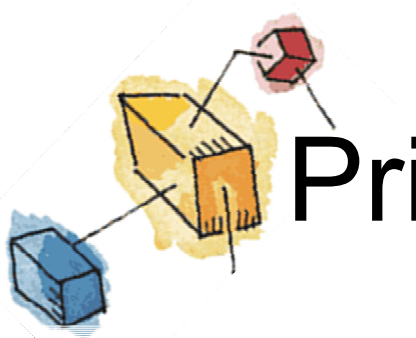


(b) Interleaving and overlapping (multiprocessing; two processors)

Blocked Running

Figure 2.12 Multiprogramming and Multiprocessing

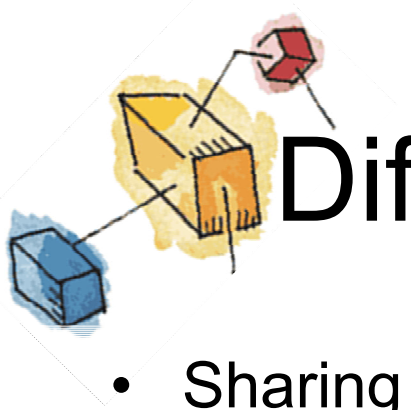




Principles of Concurrency

- **Interleaving** and **overlapping** can be viewed as examples of concurrent processing and present the same problems.
- The relative speed of execution of processes cannot be predicted.
 - depends on activities of other processes
 - the way the OS handles interrupts
 - scheduling policies of the OS

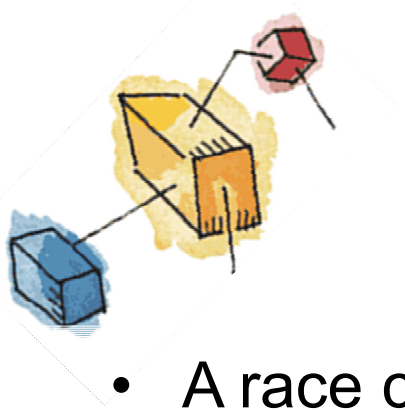




Difficulties of Concurrency

- Sharing of global resources
 - Example: global variables
- Difficult for the OS to manage the allocation of resources optimally
 - Multiple processes may request use of the same resource.
- Difficult to locate programming errors
 - Results are not deterministic and reproducible.

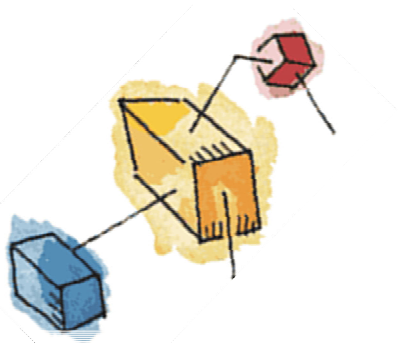




Race Condition

- A race condition occurs when multiple processes or threads read and write shared data items.
 - The final result depends on how the execution of instructions in the multiple processes interleaves.
 - The “loser” of the race is the process that updates last and will determine the final value of the variable.



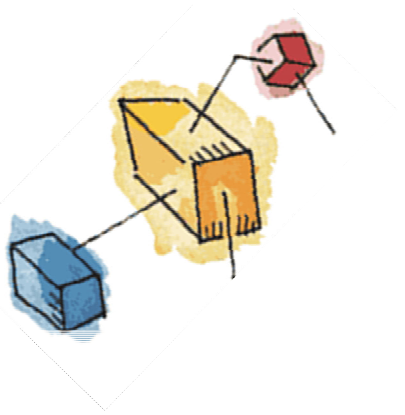


Execution Sequence Example

- Assume X is a global variable which can be accessed by P1 and P2

$X = 5000$	
P1	P2
$(X = X + 2000)$	$(X = X - 1000)$
1. $\text{Reg} \leftarrow \text{Read Mem}$ 2. $\text{Reg} \leftarrow \text{Reg} + 2000$ 3. $\text{Reg} \rightarrow \text{Write Mem}$	1. $\text{Reg} \leftarrow \text{Read Mem}$ 2. $\text{Reg} \leftarrow \text{Reg} - 1000$ 3. $\text{Reg} \rightarrow \text{Write Mem}$





Execution Sequence Example

		Reg	X
P1	1	5000	5000
P1	2	7000	5000
P1	3	7000	7000
P2	1	7000	7000
P2	2	6000	7000
P2	3	6000	6000

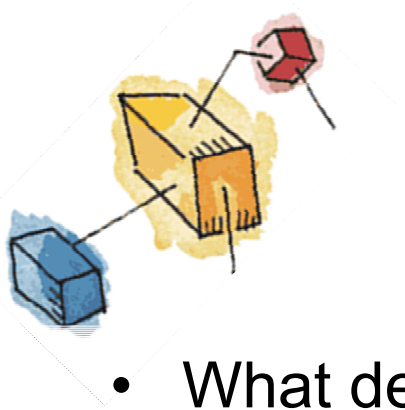
		Reg	X
P1	1	5000	5000
P2	1	5000	5000
P1	2	7000	5000
P2	2	4000	5000
P1	3	7000	7000
P2	3	4000	4000

Can you give a sequence which results in X being 7000 at the end?

		Reg	X
P1	1	5000	5000
P1	2	7000	5000
			7000

Nondeterministic result !!!

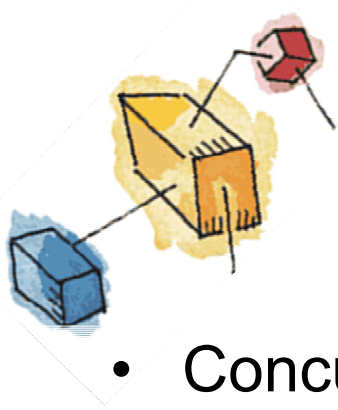




Operating System Concerns

- 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 for each active process; multiple processes want access to the same resource
 - protect the data and physical resources of each process against interference by other processes
 - **ensure that a process and its output must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes**

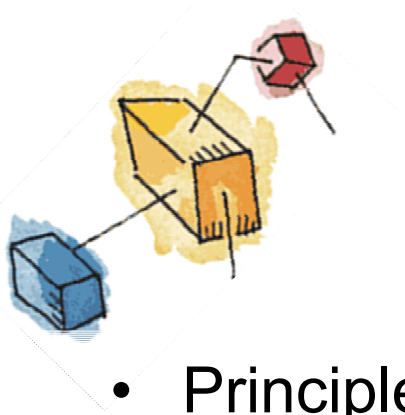




Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource such as I/O devices, memory, and processor time.
- In the case of competing processes, three control problems must be faced:
 - **Need for Mutual Exclusion**
 - Example: In the case of the printer, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.
 - Deadlock
 - Starvation





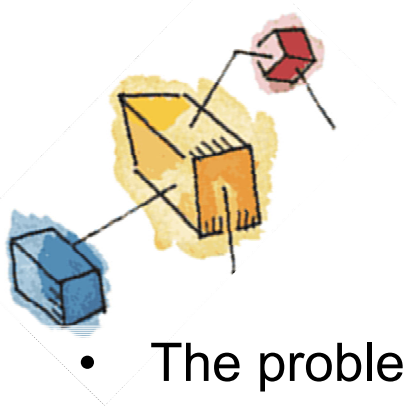
Roadmap

- Principles of Concurrency

→ Mutual Exclusion

- Semaphores
- IPC - Message Passing
- Readers/Writers Problem

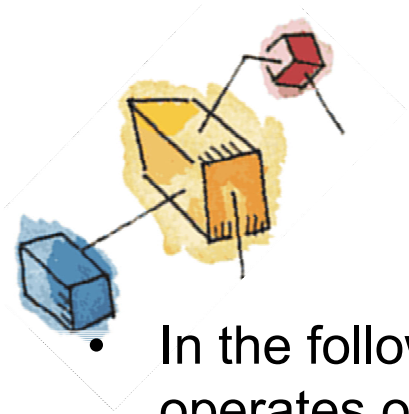




Mutual Exclusion

- The problem concerns a group of processes which need access to some resource that cannot be used simultaneously by more than one single process.
- Control of competition involves the OS because it is the OS that allocates resources.
- The processes themselves also need to be able to express the requirement for mutual exclusion.
- *Critical Section*: The piece of code within a process that accesses a shared resource (data structure or device) that must **not** be concurrently accessed by other processes.
- It is important that only one program at a time be allowed in its critical section.





Mutual Exclusion

- In the following figure, each process includes a critical section that operates on some resource Ra.
- The two functions: **entercritical** and **exitcritical** are used to enforce mutual exclusion: any process that attempts to enter its critical section while another process is in its critical section is made to wait.
- BUT, how to provide the functions entercritical and exitcritical?

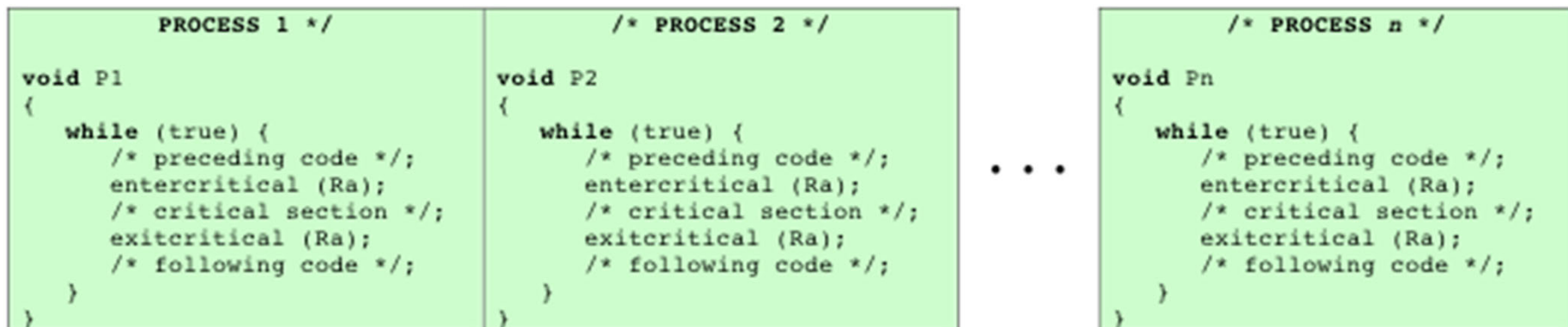
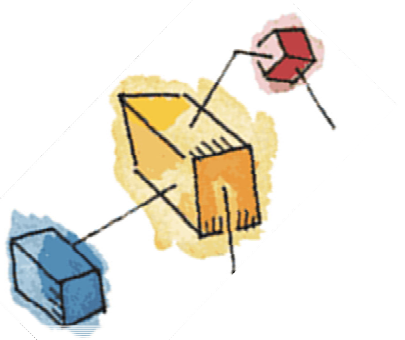


Figure 5.4 Illustration of Mutual Exclusion

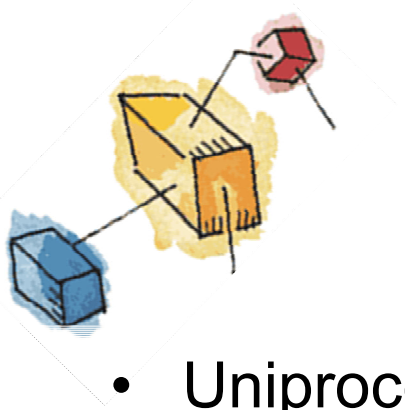




Requirements for Mutual Exclusion

- Any facility that is to provide support for mutual exclusion should meet the following requirements:
 - Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource
 - A process that halts in its noncritical section must do so without interfering with other processes
 - No deadlock or starvation
 - When no process is in a critical section, any process that request entry to its critical section must be permitted to enter without delay
 - No assumptions are made about relative process speeds or number of processes
 - A process remains inside its critical section for a finite time only





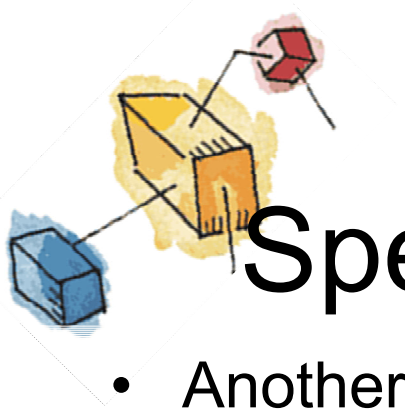
Hardware Support: Disabling Interrupts

- Uniprocessors only allow interleaving, no overlapping.
- To guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted while it is in the critical section.

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

- Will it work for multi-processor architecture?





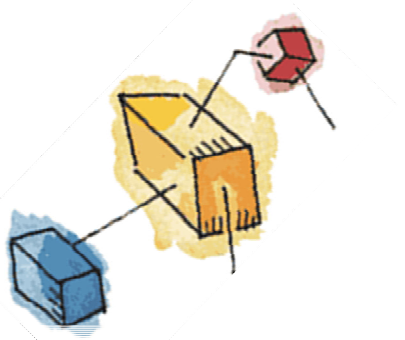
Hardware Support: Special Machine Instructions

- Another hardware solution:
 - Compare-and-Swap (CAS) Instruction

atomic CPU instruction
that cannot be interrupted

- A **compare** is made between a memory value and a test value.
- If the values are the same, a **swap** occurs.





Compare & Swap Instruction

Pesudo Code

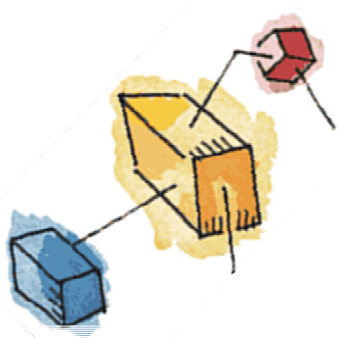
```
int compare_and_swap (int *word,
                      int testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval)
        *word = newval;
    return oldval;
}
```

Only one process finding **bolt** equal to 0 will enter its critical section; all other processes go into a **busy waiting** mode

Usage Example

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

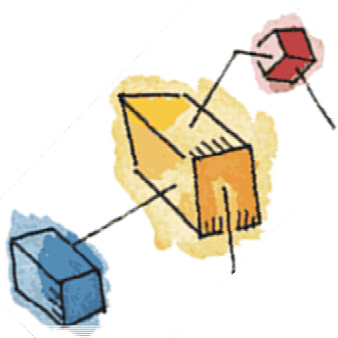
(a) Compare and swap instruction



Special Machine Instruction

- Advantages
 - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
 - Simple and easy to verify
 - It can be used to support multiple critical sections; each critical section can be defined by its own variable

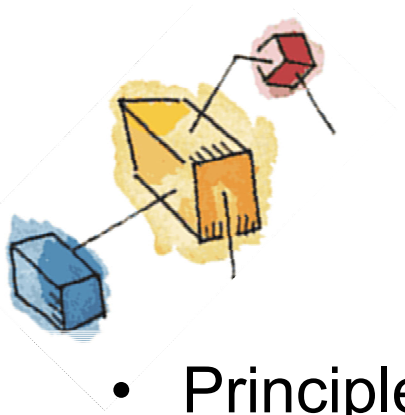




Special Machine Instruction

- Disadvantages
 - *Busy-waiting* consumes processor time
 - Starvation is possible
 - When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary; some process could indefinitely be denied access.
 - Deadlock is possible
 - P1 enters its critical section and is then preempted by a higher-priority P2.
 - P2 attempts to use the same resource as P1 but is denied access because of the mutual exclusion mechanism.
 - P2 goes into a busy waiting loop.
 - The lower-priority P1 will never be dispatched.





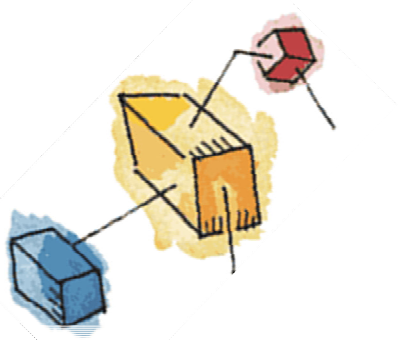
Roadmap

- Principles of Concurrency
- Mutual Exclusion

→ Semaphores

- IPC - Message Passing
- Readers/Writers Problem

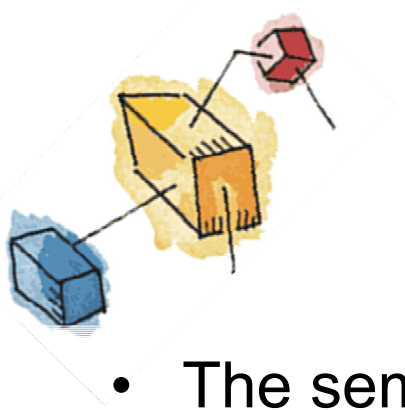




Semaphore

- Fundamental principle: multiple processes can cooperate by means of simple signals such that a process can be forced to stop at a specified place until it has received a specific signal.
- Semaphore: an integer value used for signalling among processes.
- Three operations on a semaphore are all *atomic*:
 - Initialize to a non-negative integer value
 - **semWait** decrements the semaphore value: to receive a signal
 - **semSignal** increments the semaphore value: to transmit a signal



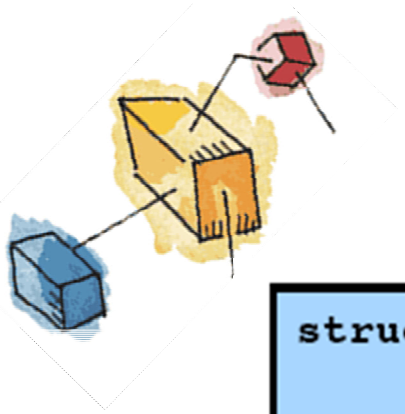


Semaphore

- The semaphore is initialized to zero or a positive value.
- When the value is positive, that value equals the number of processes that can issue a **wait** and immediately continue to execute.
- When the value is zero, the next process to issue a **wait** is blocked, and the semaphore value goes negative.
 - Each subsequent **wait** further decrements the value.
 - The negative value equals the number of processes waiting to be unblocked.
- Each **signal** unblocks one of the waiting processes, if any.



General 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 */;
    }
}
```

-ve value equals the number of blocked processes



Figure 5.3 A Definition of Semaphore Primitives

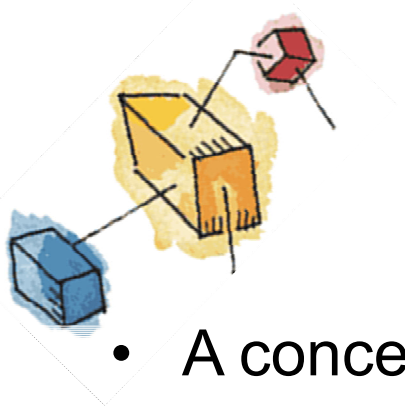


Binary Semaphore Primitives

```
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 */;
    }
}
```

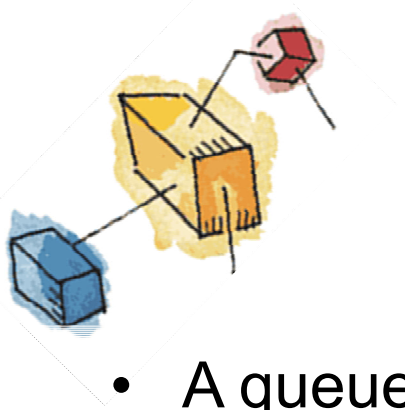
Figure 5.4 A Definition of Binary Semaphore Primitives

Mutex



- A concept related to the binary semaphore is the **mutex**.
- Two operations: **lock()** and **unlock()**
- When a process tries to lock a mutex
 - It may acquire the lock if the mutex is not being locked
 - Or it may fail and enter in a waiting mode if the mutex is being locked
- When a process tries to unlock a mutex
 - It releases the lock and makes the mutex available for another process to lock
- A key difference between binary semaphore and mutex is that the process that locks the mutex must be the one to unlock it, i.e., a process **cannot** unlock a mutex that is not locked by itself.

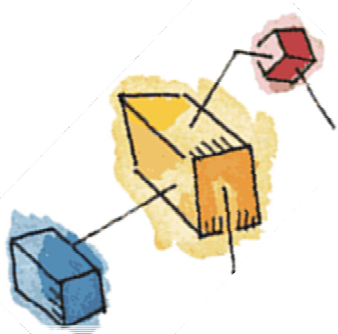




Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore.
 - In what order are processes removed from the queue?
- **Strong Semaphores** use FIFO: the process that has been blocked the longest is released from the queue first.
- **Weak Semaphores** do not specify the order of removal from the queue.





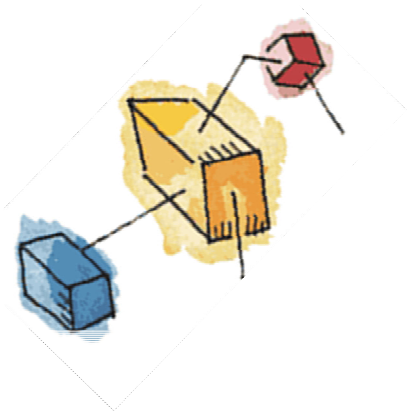
Mutual Exclusion Using Semaphores

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

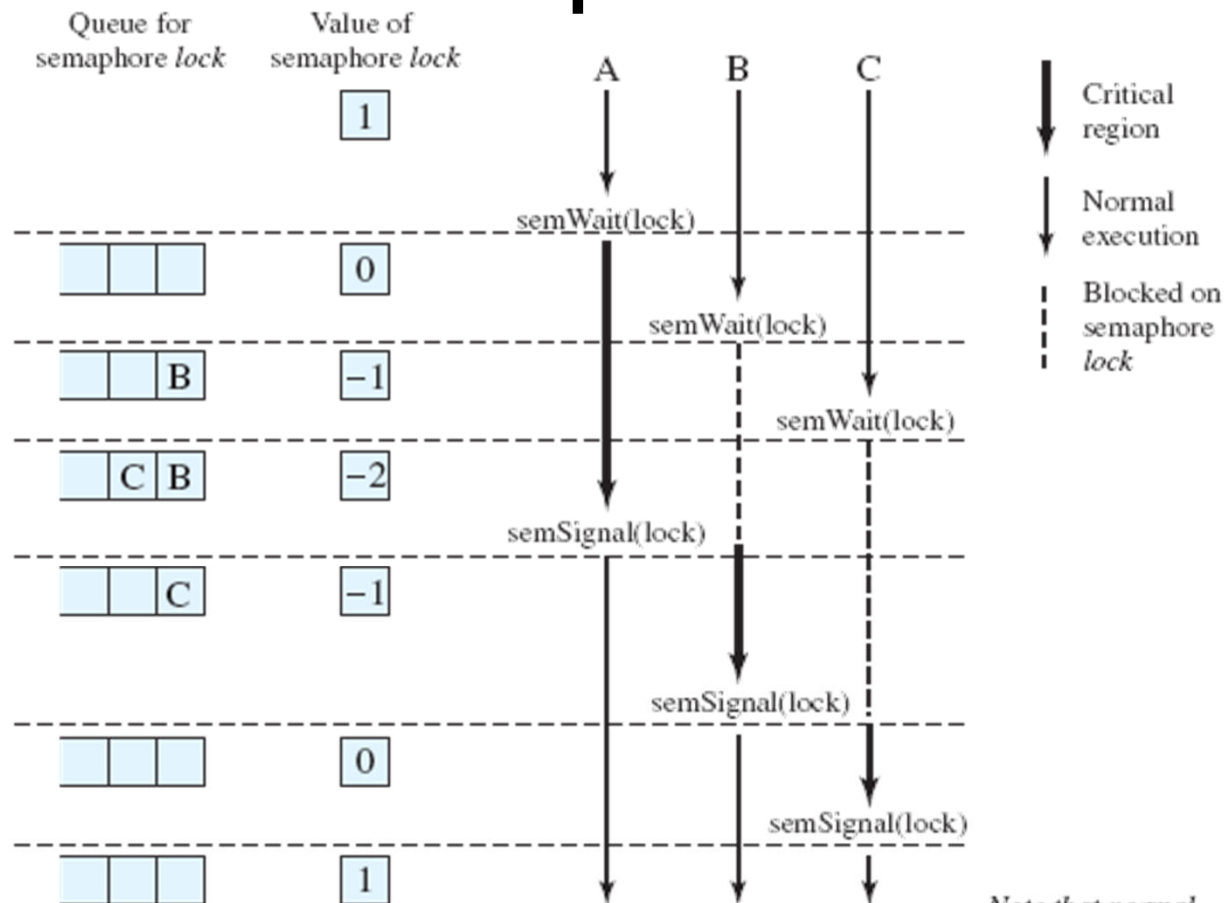
1. The first process will be able to enter the critical section immediately, setting the value of *s* to 0.
2. Any other processes attempting to enter the critical section will be blocked, *s* is decremented.
3. When the first process departs, *s* is incremented and one of the blocked processes (if any) becomes Ready.



Figure 5.6 Mutual Exclusion Using Semaphores



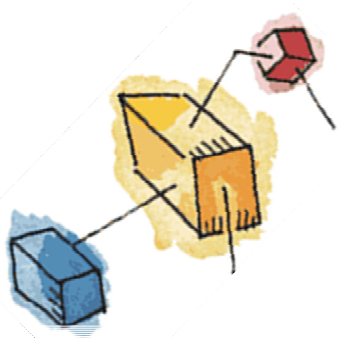
Processes Using Semaphore



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



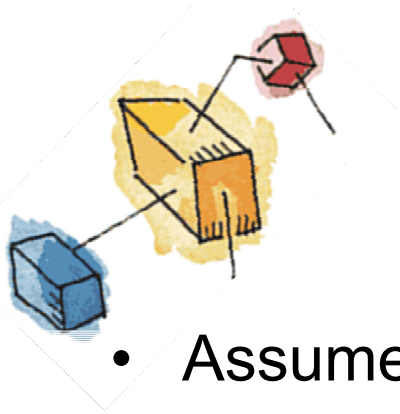
Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore



Producer/Consumer Problem

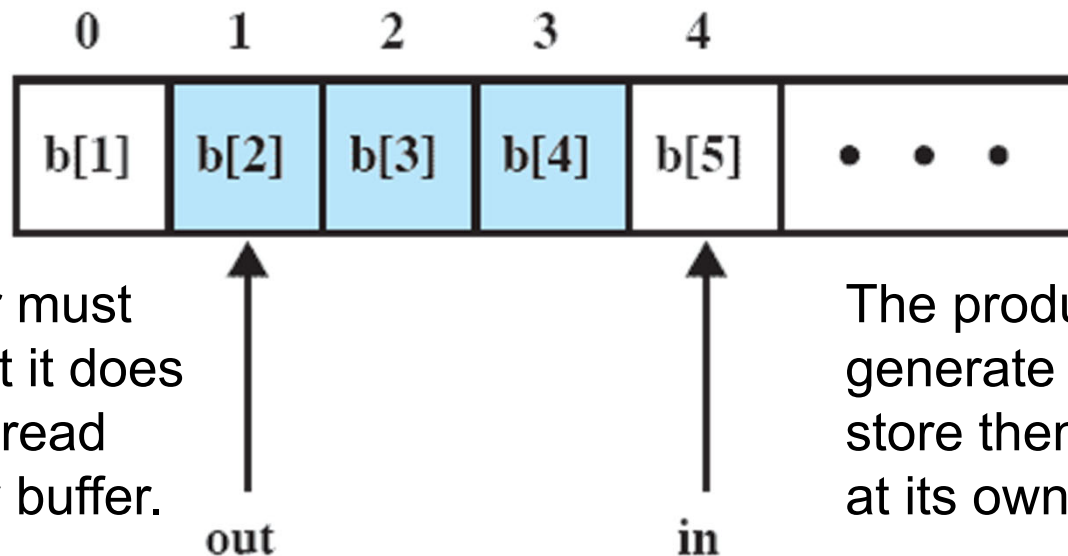
- General Statement:
 - One or more producers are generating data and placing these in a buffer.
 - A single consumer is taking items out of the buffer one at time.
 - Only one producer or consumer may access the buffer at any one time.
- The Problem:
 - Ensure that the Producer can't add data into full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





Infinite Buffer

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



The consumer must make sure that it does not attempt to read from an empty buffer.

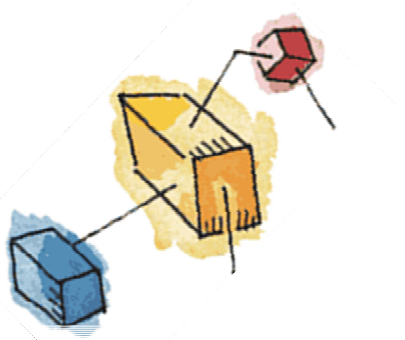
The producer can generate items and store them in the buffer at its own pace.

Note: shaded area indicates portion of buffer that is occupied



Figure 5.8 Infinite Buffer for the Producer/Consumer Problem





Idea

- A consumer is
 - blocked when removing from empty buffer (i.e., ZERO item in the buffer)
 - unblocked when an item is inserted by the producer (i.e., >0 number of items in the buffer)
- A semaphore, n = number of items in the buffer, can be used to implement the idea.



Semaphores

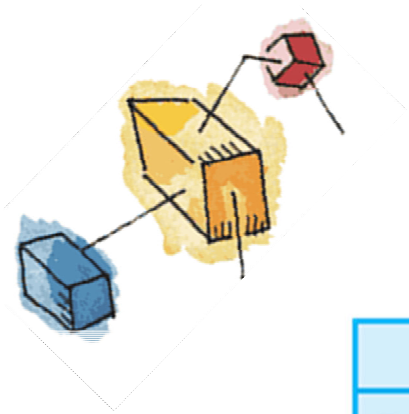
```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
  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

n is equal to the number of items in the buffer.

Prevent the consumer and any other producer from accessing the buffer during the append operation.

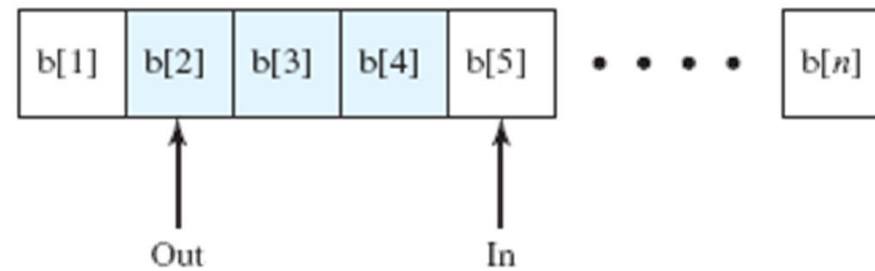
The consumer must wait on both semaphores before proceeding.

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

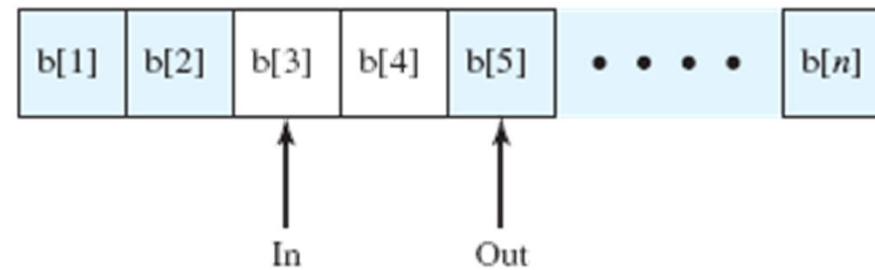


Bounded Buffer

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



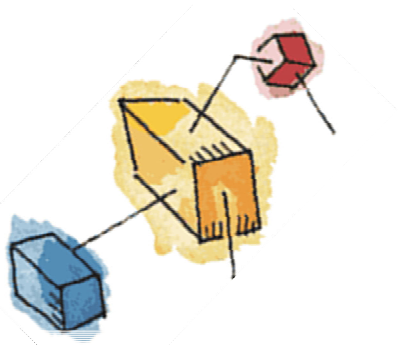
(a)



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem



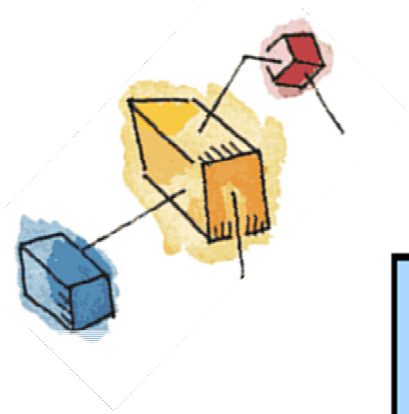


Idea

- A producer is
 - blocked when inserting in full buffer (i.e., ZERO empty space in the buffer)
 - unblocked when an item is removed by the consumer (i.e., >0 number of empty space in the buffer)
- A semaphore, e = number of empty spaces in the buffer, can be used to implement the idea.



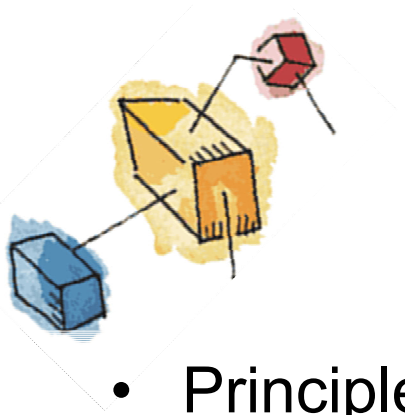
Semaphores



```
/* 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);
}
```

e keeps track of the number of empty spaces





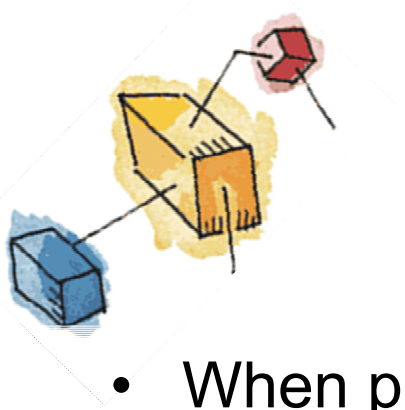
Roadmap

- Principles of Concurrency
- Mutual Exclusion
- Semaphores

→ IPC - Message Passing

- Readers/Writers Problem

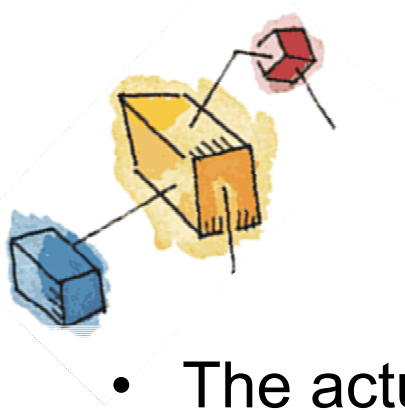




Message Passing

- When processes interact with one another, they may need to satisfy two fundamental requirements:
 - synchronization: to enforce mutual exclusion
 - communication: to exchange information
- Message Passing is one approach to providing both of these functions.

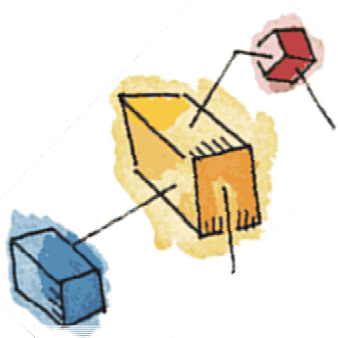




Message Passing

- The actual function is normally provided in the form of a pair of primitives:
 - `send (destination, message)`
 - `receive (source, message)`
- If P1 and P2 wish to communicate, they must send messages to and receive messages from each other.
 - a ***communication link*** must exist between them

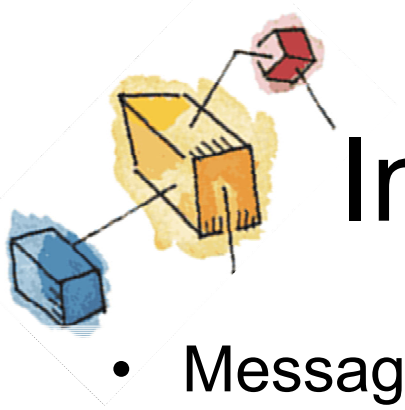




Direct Communication

- Processes must name each other explicitly:
 - **send**(*P1*, *message*) – send a message to process P1
 - **receive**(*P2*, *message*) – receive a message from process P2
- Properties of communication link
 - The processes need to know each other's identity to communicate.
 - A link is associated with exactly two processes.
 - Between each pair of processes, there exists exactly one link.

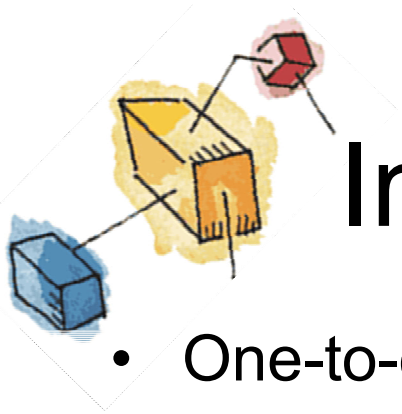




Indirect Communication

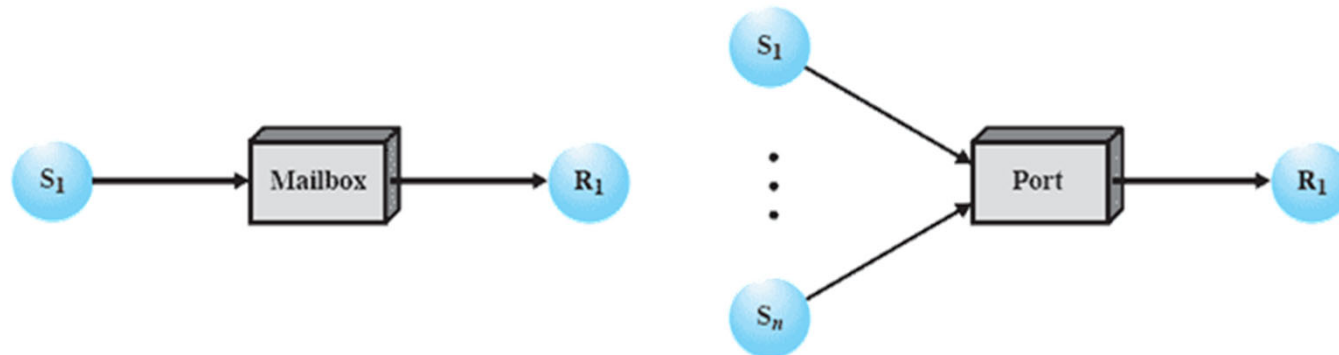
- Messages are directed to and received from **mailboxes** (also referred to as **ports**)
 - `boxid = mailbox-create(IDENTIFIER)`
 - `send(boxid, message)`
 - `receive(boxid, message)`
- One process sends a message to the mailbox and the other process picks up the message from the mailbox
- Properties of communication link
 - A link is established between a pair of processes only if they have a shared mailbox.
 - A link may be associated with many processes.
 - Each pair of processes may have several links, each link corresponds to one mailbox.





Indirect Communication

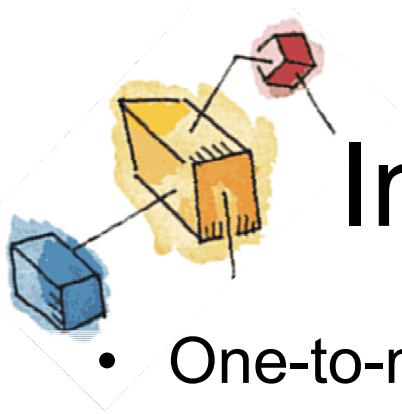
- One-to-one relationship
 - allows a private communications link to be set up between two processes.
- Many-to-one relationship
 - useful for client/server interaction, one process provides service to a number of other processes.
 - the mailbox is often referred to as a port.



(a) One to one

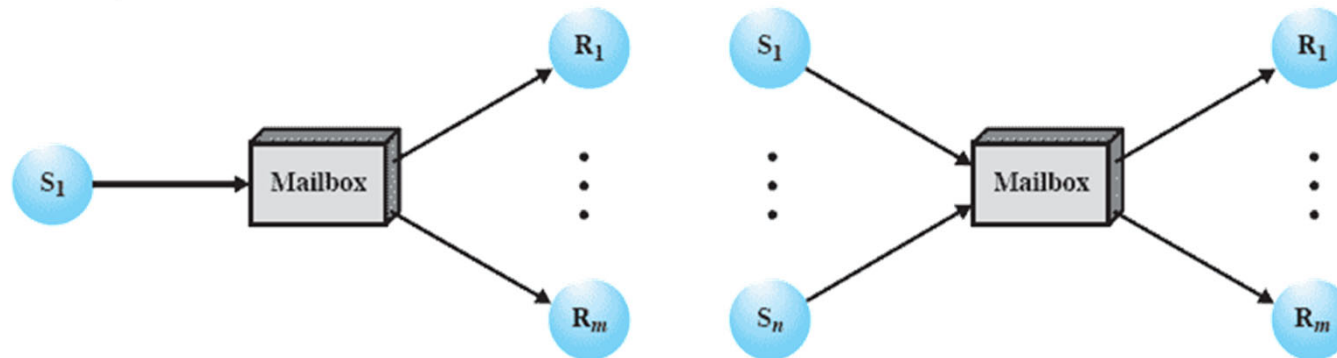
(b) Many to one





Indirect Communication

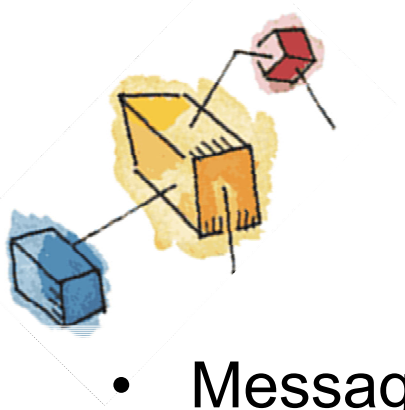
- One-to-many relationship
 - allows for one sender and multiple receivers;
 - useful for applications where a message is to be broadcast to a set of processes.
- Many-to-many relationship
 - allows multiple server processes to provide concurrent service to multiple clients.



(c) One to many

(d) Many to many

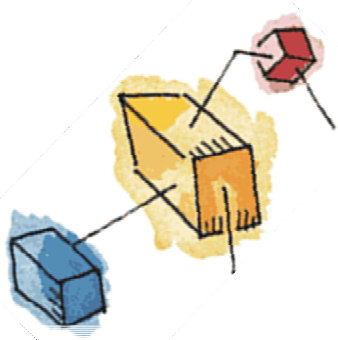




Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send**: the sender is blocked until the message is received
 - **Blocking receive**: the receiver is blocked until a message arrives
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send**: the sender sends the message and continues
 - **Non-blocking receive**: the receiver receives the message or abandons the attempt to receive and continues





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

non-blocking send

A process wishing to enter its critical section attempts to receive a message.

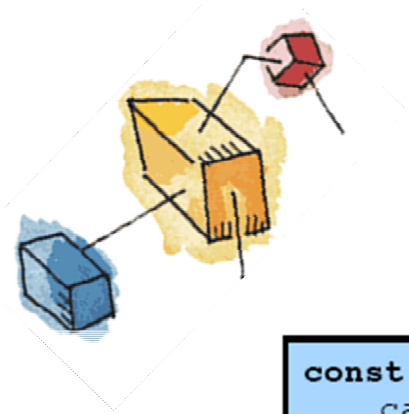
The process places the message back into the mailbox.

The message functions as a token that is passed from process to process.



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);
}
```

blocking receive

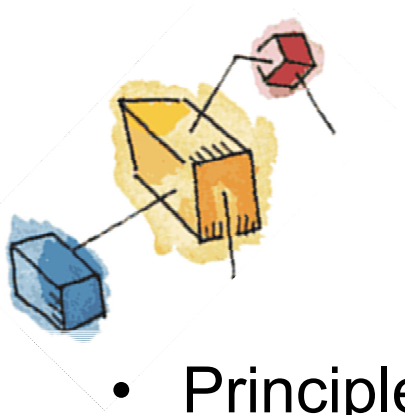
non-blocking send

Data are sent as messages to **mayconsume** which serves as the buffer.

The number of messages in **mayproduce** shrinks with each production and grows with each consumption.

mayproduce is filled with a number of null messages equal to the capacity of the buffer.



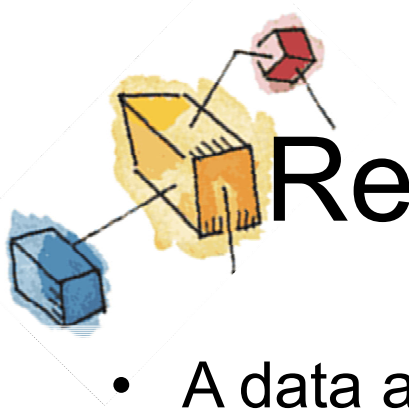


Roadmap

- Principles of Concurrency
- Mutual Exclusion
- Semaphores
- IPC - Message Passing

➔ Readers/Writers Problem

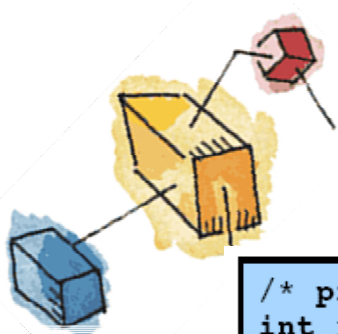




Readers / Writers Problem

- A data area is shared among many processes
 - Some processes only read the data area (readers), some only write to the data area (writers)
- Conditions to satisfy:
 1. Any number of readers may simultaneously read the file.
 2. Only one writer at a time may write to the file.
 3. 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);
}
```

readcount keeps track of the number of readers

x assures that **readcount** is updated properly

the **first** reader that attempts to read should wait on **wsem**

wsem is used to enforce mutual exclusion: as long as one writer is accessing the shared data area, no other writers and no readers may access it

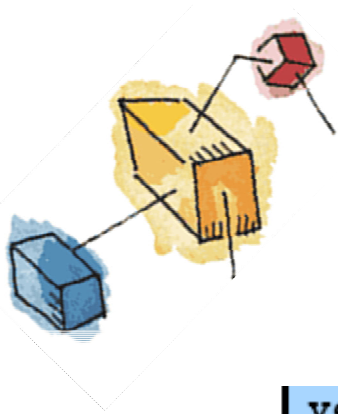




Readers/Writers Problem

- Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader reading.
 - Therefore, writers are subject to **starvation**
- An alternative solution: no new readers are allowed access to the data area once at least one writer wants to write





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);
    }
}

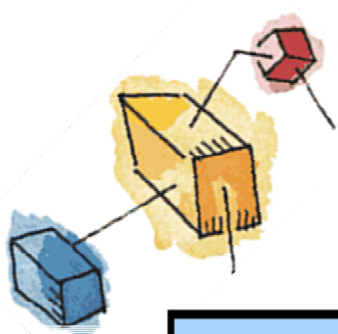
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

y controls the updating of
writecount

writecount controls the setting
of **rsem**

rsem inhibits all readers while
there is at least **one** writer
desiring access to the data area





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);
    }
}
```

only one reader is allowed to queue on **rsem**, with any additional readers queuing on **z**



Reader

```
semWait (rsem);

semWait(x);
readcount++;
if(readcount==1)
    semWait(wsem);
semSignal(x);

semSignal (rsem);
READUNIT( );

semWait(x);
readcount--;
if (readcount==0)
    semSignal(wsem);
semSignal(x);
```

Unblock the following readers.

Reader

blocked

```
semWait (rsem);

semWait(x);
readcount++;
if(readcount==1)
    semWait(wsem);
semSignal(x);

semSignal (rsem);
READUNIT( );

semWait(x);
readcount--;
if (readcount==0)
    semSignal(wsem);
semSignal(x);
```

Unblock the writer.
The writer is then blocked on **wsem**

Reader

blocked

```
semWait (rsem);

semWait(x);
readcount++;
if(readcount==1)
    semWait(wsem);
semSignal(x);

semSignal (rsem);
READUNIT( );

semWait(x);
readcount--;
if (readcount==0)
    semSignal(wsem);
semSignal(x);
```

Writer

blocked

```
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);
```

The writer has to wait until the all readers finish reading and **semSignal(wsem)**.



Reader

```
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);
```

Unblock the writer.
The writer is then
blocked on **wsem**

Reader

blocked

```
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);
```

Unblock the 2nd reader
waiting on **z**, but it will be
blocked on **rsem**.

Reader

blocked

```
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);
```

Unblock the writer

Writer

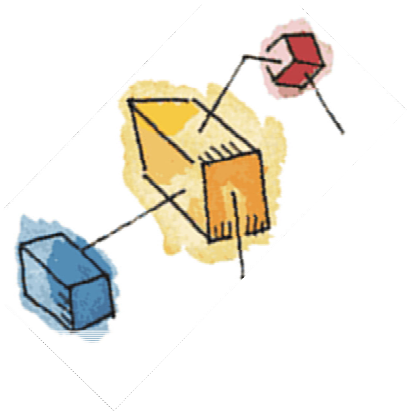
blocked

```
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);
```

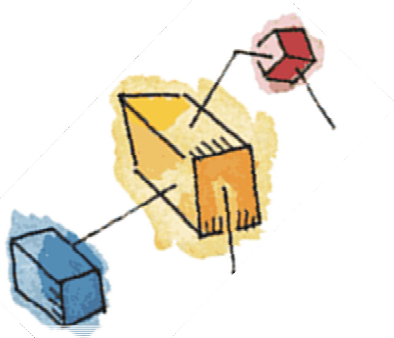


Writers have Priority

Table 5.6 State of the Process Queues for Program of Figure 5.23

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





Writers Have Priority Message Passing

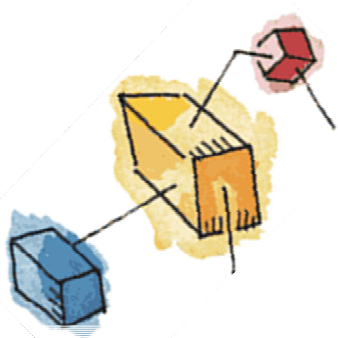
```
void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT ();
        rmsg = i;
        send (finished, rmsg);
    }
}

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++;
        }
    }
}
```

count = 100, initially

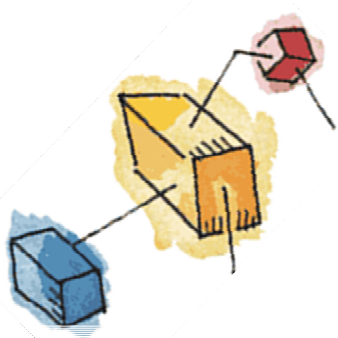




Writers Have Priority Message Passing

- Processes wishing to access the data area send a request message to the controller
 - Granted access with an “OK” reply message
 - Completion of access with a “finished” message
- **count** is initialized to 100 ($>$ the maximum possible number of readers)
- **count** $>$ 0: no writer is waiting, clear active readers first, then service write requests and then read requests
- **count** = 0: the only request outstanding is a write request
- **count** $<$ 0: a writer has made a request and is being made to wait to clear all active readers

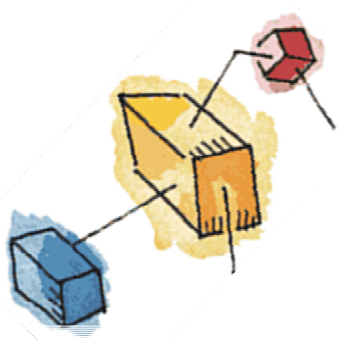




Writers Have Priority Message Passing

- Writers have priority
 - The controller services write requests before read requests.
- Readers only and readers can read simultaneously
 - After receiving a read request, the controller sends “OK” and does not block to receive “finished” ($\text{count} > 0$) → can continue to receive other read requests.
- Writers only and only one writer can write at a time
 - After receiving a write request, the controller sends “OK” and blocks to receive “finished” ($\text{count} == 0$) → cannot continue until the writer finishes.





Writers Have Priority Message Passing

- Both readers and writers and read first (i.e., a writer arrives while the first reader is reading)
 - The controller receives the write request and makes the writer to wait until receiving “finished” from the first reader ($\text{count} < 0$).
 - When the first reader finishes, the controller sends “OK” to the writer and blocks to receive “finished” ($\text{count} == 0$).
- Both readers and writers and write first (i.e., a reader arrives while the first writer is writing)
 - The controller blocks to receive “finished” while the first writer is writing ($\text{count} == 0$).
 - When the first writer finishes, the controller receives the read request and send “OK” to the reader” ($\text{count} > 0$).

