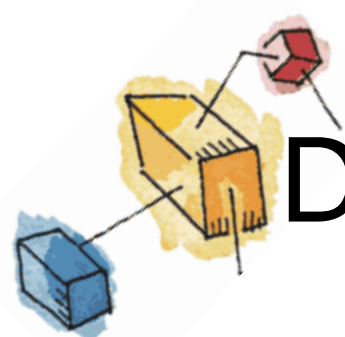# Chapter 5
# Concurrency: Mutual Exclusion and Synchronization

# 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

# Difficulties of Concurrency

- **Sharing of global resources**
  - If P1 and P2 are sharing common variable, then care should be taken with read and write operations

- **Optimally managing the allocation of resources**
  - If P1 is given access to I/O channel and it gets suspended before using I/O, then performance will suffer

- **Difficult to locate programming errors as results are not deterministic and reproducible**
  - Due to overwriting of values
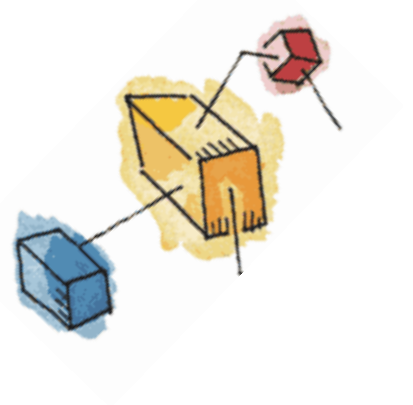
# A Simple Example

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

- Consider that we have a single-processor multiprogramming system supporting a single user.

  - The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output.

# A Simple Example

- Each application needs to use the procedure echo,

  - So it makes sense for it to be a shared procedure that is loaded into a portion of memory global to all applications.

  - Thus, only a single copy of the echo procedure is used, saving space.

# A Simple Example: Single Processor, Multiprogramming System

- P1 calls echo, but interrupted after first line (chin=x)
- P2 activated, calls echo and runs till end (chin=chout=y)
- P1 resumes, but the value is overwritten (prints y!!)

- **Problem:**
  - Shared global variables / procedure

- **Solution:**
  - Permit only one process to access the procedure at a time (works for multiprocesssing systems as well)

# A Simple Example: On a Multiprocessor

**Process P1**

.

chin = getchar();

.

chout = chin;
putchar(chout);

.

.

**Process P2**

.

.

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
  - P1 completes execution
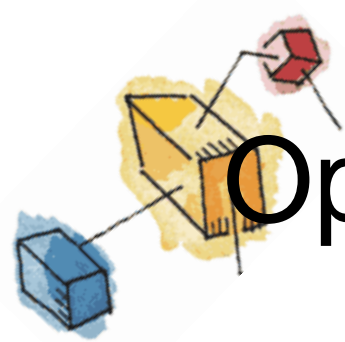  - P2 resumes and executes echo

# Race Condition

- A **race condition** occurs when
  - "Multiple processes or threads read and write data items hence the final result depends on the order of execution of the processes"
  - The output depends on who finishes the race last.
- E.g. Consider that a global variable "a" is accessed by Processes P1 and P2
  - P1 writes a=1
  - P2 writes a=2
  - The loser wins!

# 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
  - Manage concurrency properly

# Process Interaction Scenarios

- Can be divided in three categories:

  - Processes unaware of each other
    - Competition
  - Processes indirectly aware of each other
    - Co-operation by sharing
  - Processes directly aware of each other
    - Co-operation by communication

# Process Interaction: Unaware

- Example: Multiprogramming
  - Each process has different goal
- OS needs to be concerned about the competition for resources
  - P1 and P2 may compete for Printer

- **Potential Control Problems:**
  - Mutual Exclusion
  - Deadlock
  - Starvation

# Process Interaction: Unaware

- **Mutual Exclusion:**
  - If P1 and P2 both need printer
    - Mutual exclusion should be enforced
  - Only one process should be allowed to use printer at a time
- Critical Resource:
  - Printer can be called a critical resource for this case
- Critical Section:
  - The part of code where printer is being accessed can be called critical section

# Process Interaction: Unaware

- **Deadlock:**
  - Occurs due to enforcement of mutual exclusion

  - Consider two processes (P1,P2) and two resources (R1,R2)
  - Process P1 is allocated R1 and P2 is allocated R2
  - P1 requires R2 to complete and P2 requires R1 to complete --- Deadlock!

# Process Interaction: Unaware

- **Starvation:**
  - Occurs due to enforcement of mutual exclusion

  - Consider that three processes P1,P2 and P3 need periodic access to resource R
  - If P1 is allotted R --- P2, P3 get delayed
  - When P1 exits critical section, either P2 or P3 will be allowed to enter critical section
    - Suppose P3 is granted access
    - After P3, if P1 gets access to R again, P2 might suffer from starvation

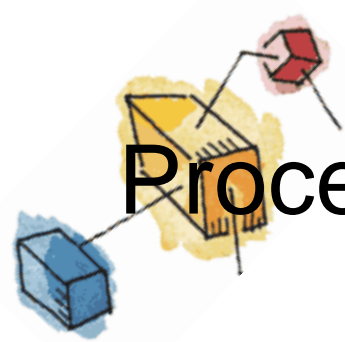# Process Interaction: Unaware

- Hence, the general structure for each process should be as follows:

```
void P1 {
  while(true) {
          //Preceding Code
          enterCritical(Ra);
          //critical section
          exitcritical(Ra);
          //following Code
      }
}
```

# Process Interaction: Indirect Awareness

- Multiple processes may have access to shared variables / files / databases

- Integrity needs to be ensured

- Data is held on resources e.g. memory or other storage
  - Same control problems occur – Mutual Exclusion, Deadlock, Starvation
  - Difference here is we need only write operations to be mutually exclusive
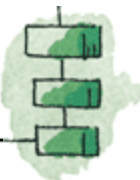
# Process Interaction: Indirect Awareness

- One more requirement: Data Coherence
- Consider the code given which has to make sure that a=b. (a and b are initialized with same value)

```
P1:
        a = a + 1;
        b = b + 1;
P2:
        b = 2 * b;
        a = 2 * a;
```
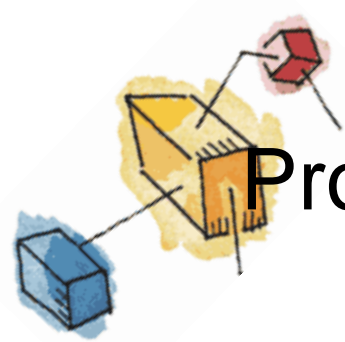
# Process Interaction: Indirect Awareness

- Assume that mutual exclusion is done on a and b (If P1 is updating a, P2 can not update a)

- If the processes execute as given below, then a=b will not hold

$$a = a + 1;$$
$$b = 2 * b;$$
$$b = b + 1;$$
$$a = 2 * a;$$

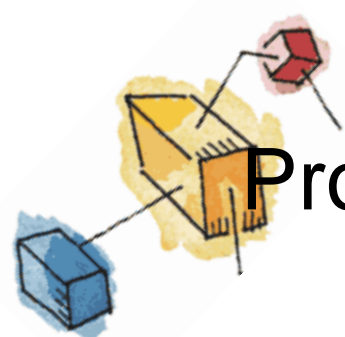- So we need to put entire execution sequence in critical section

# Process Interaction: Direct Awareness

- Processes co-operate by communication

- When processes cooperate by communication, they participate in a common effort that links all of the processes.

- The communication provides a way to synchronize, or coordinate, the various activities.

# Process Interaction: Direct Awareness

- Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation.

- However, the problems of deadlock (P1 & P2 waiting for messages from each other) and starvation (P1 and P2 keep communicating, starving P3) are still present.

# 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 | • Results of one process independent of the action of others<br>• Timing of process may be affected | • Mutual exclusion<br>• Deadlock (renewable resource)<br>• Starvation |
| Processes indirectly aware of each other (e.g., shared object) | Cooperation by sharing | • Results of one process may depend on information obtained from others<br>• Timing of process may be affected | • Mutual exclusion<br>• Deadlock (renewable resource)<br>• Starvation<br>• Data coherence |
| Processes directly aware of each other (have communication primitives available to them) | Cooperation by communication | • Results of one process may depend on information obtained from others<br>• Timing of process may be affected | • Deadlock (consumable resource)<br>• 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

- 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
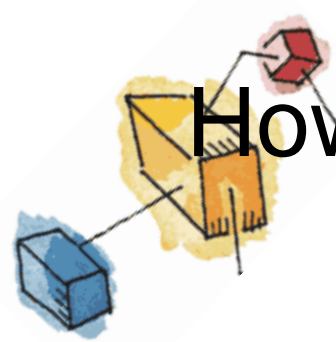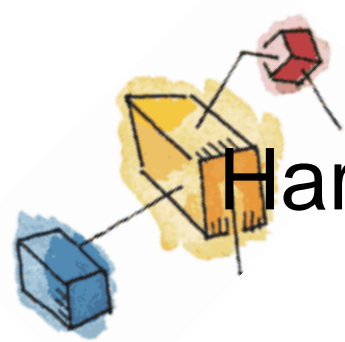
# How to achieve the Requirements of Mutual Exclusion?

- Three ways possible

  1. Leave the responsibility to the processes
     - High overhead
  2. Use special purpose machine instructions (hardware support)
     - Reduces overhead
  3. Support of OS
     - Most popular

# Hardware Support for Mutual Exclusion

- Two methods:

  1. Interrupt Disabling

  2. Special Machine Instructions

# Disabling Interrupts

- Uniprocessors only allow interleaving

- A process runs until it invokes an operating system service or until it is interrupted

- Disabling interrupts guarantees mutual exclusion

- **Issues:**
  - Execution efficiency is degraded as interleaving gets limited.
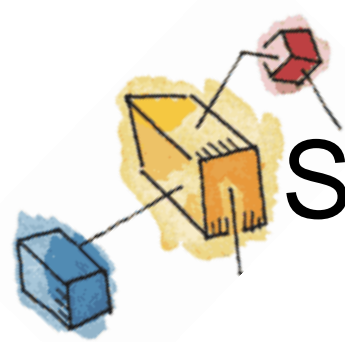  - Will not work in multiprocessor architecture

# Pseudo-Code

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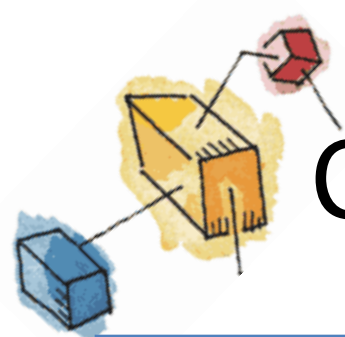
# Special Machine Instructions

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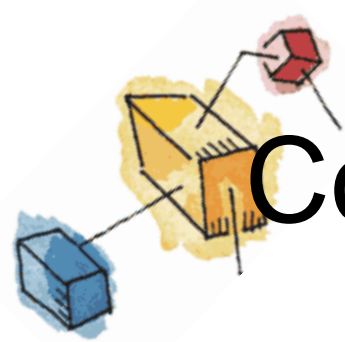
- Exchange Instruction

# Compare&Swap Instruction

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

# Compare&Swap Instruction

- This version of the instruction checks a memory location (*word) against a test value (testval).

- If the memory location's current value is testval, it is replaced with newval;
  - otherwise it is left unchanged.

- The old memory value is always returned;
  - thus, the memory location has been updated if the returned value is the same as the test value.

- This is an atomic instruction

# 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));

}
```
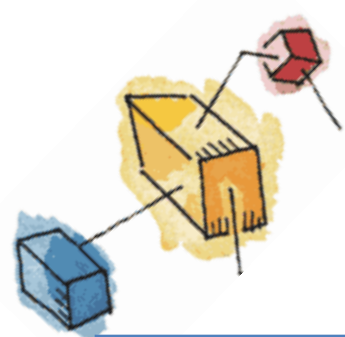
(a) Compare and swap instruction

# Mutual Exclusion

- Bolt is initialized to 0
- First call (P1) would be C&S(0,0,1)
  - Word and testval same so word gets updated
  - Return value is 0, hence P1 can enter critical section (while condition false)

- P2 calls C&S(1,0,1)
  - No change
  - Return value is 1, hence P2 waits (while condition true)

# Exchange instruction

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

- Exchange the contents of register with the content of memory location
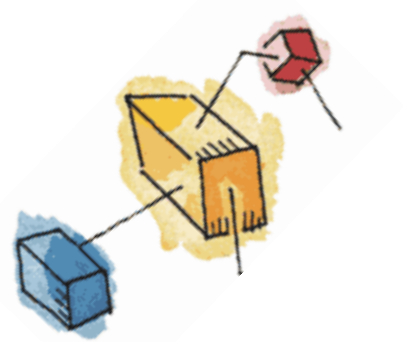
# Mutual Exclusion

```
    /* 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));
}
```
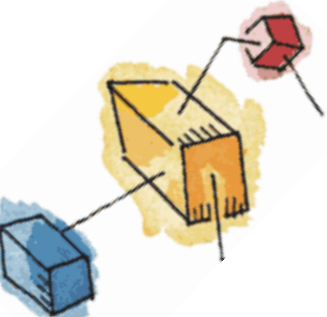
(b) Exchange instruction

# Mutual Exclusion

- For process P1,
  - keyi=1, exchange(1,0) is called
  - keyi=0, bolt=1
  - P1 enters critical section

- For process P2,
  - keyi=1, exchange(1,1) is called
  - No change, P2 waits

# 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**
  - P1 runs and calls C&S
  - Interrupted due to high priority process P2
  - P2 runs but needs the resource occupied by P1

# Semaphore

- OS and Programming language based mechanism
- Two or more processes co-operate using signals
  - A process has to stop at a place until it receives a signal
  - Special variables – Semaphores are used for this purpose
- Operations on Semaphores (all of which are atomic):
  - initialize,
  - Decrement (`semWait`)
  - increment. (`semSignal`)

# Operations on Semaphores

- A semaphore may be initialized to a nonnegative integer value.

- The semWait operation decrements the semaphore value.

  - If the value becomes negative, then the process executing the semWait is blocked.

  - Otherwise, the process continues execution.

# Operations on Semaphores

- The semSignal operation increments the semaphore value.

  - If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

- This type of semaphores are called Counting / General semaphores
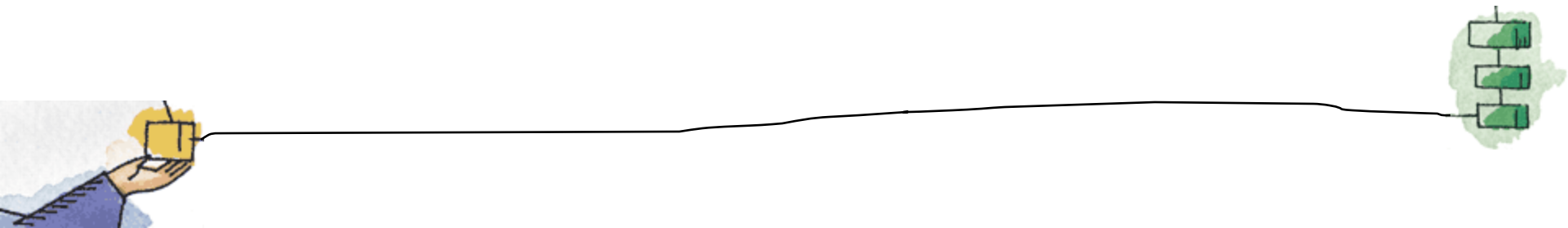
# Semaphore Primitives

```
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 */;
    }
}
```
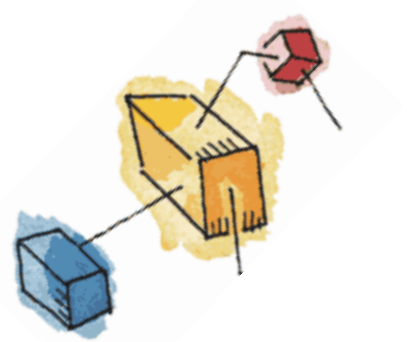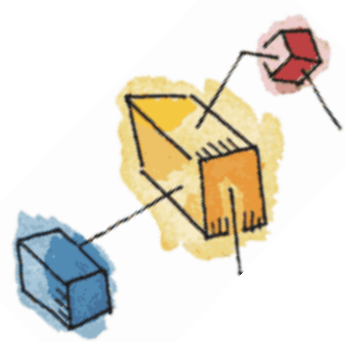
**Figure 5.3 A Definition of Semaphore Primitives**

# Binary Semaphores

- Restricted version of counting semaphores

- Operations on Semaphores (all of which are atomic):

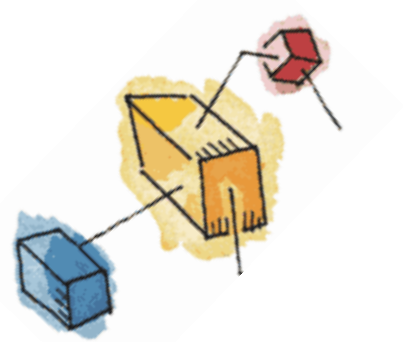    - Initialize
    - Decrement (`semWaitB`)
    - increment. (`semSignalB`)

# Binary Semaphores

- A binary semaphore may be initialized to either zero or one.

- The semWaitB operation checks the semaphore value
    - If the value is zero, process is blocked
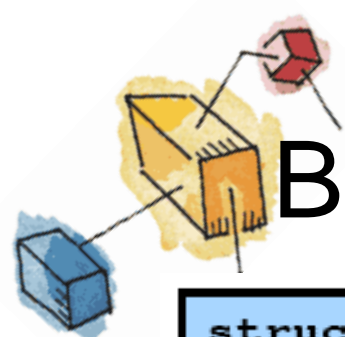    - Otherwise, value is changed to zero and the process continues execution.
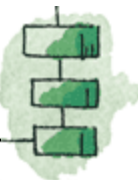
# Binary Semaphores

- The semSignalB operation checks if any processes are blocked (semaphore value zero)

  - If blocked, one process is unblocked

  - If no processes are blocked, value is set to One.

# 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
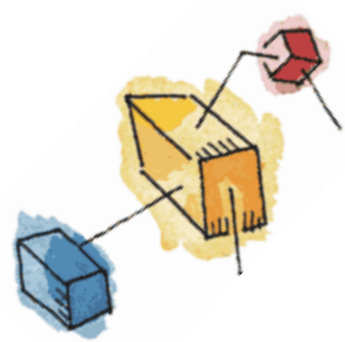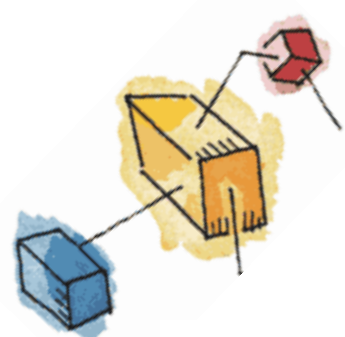
# Mutual Exclusion Using Semaphores

```
/* 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**

# Mutual Exclusion Using Semaphores

- All processes need the same resource, hence each process contains a critical section to access the resource

- First process executing semWait will be allowed to enter critical section, rest of the processes will be blocked

- When a process exits critical section, semaphore value is incremented, so any one blocked process will be allowed to enter critical section

- Can the same solution work for multiple critical sections?

# Mutual Exclusion Using Semaphores

- If the value of s is positive,
  - Equal to number of processes that can issue semWait and access critical section

- If the value of s is zero,
  - Next process that issues semWait will be blocked

- If the value of s is negative,
  - Equal to the number of processes waiting

# 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

# Buffer
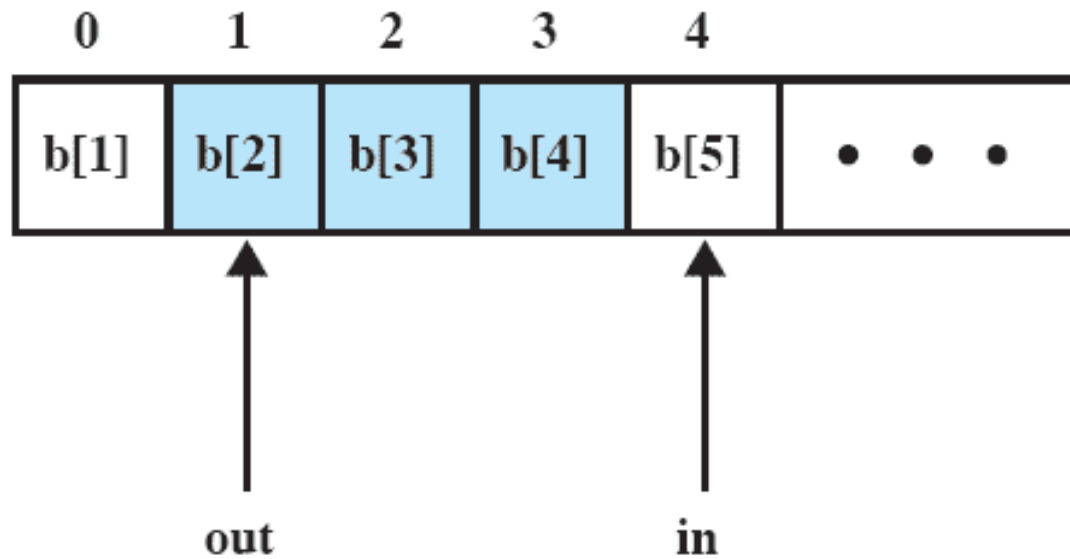


Note: shaded area indicates portion of buffer that is occupied

Figure 5.8   Infinite Buffer for the Producer/Consumer Problem

# Functions

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

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

# 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);
}
```
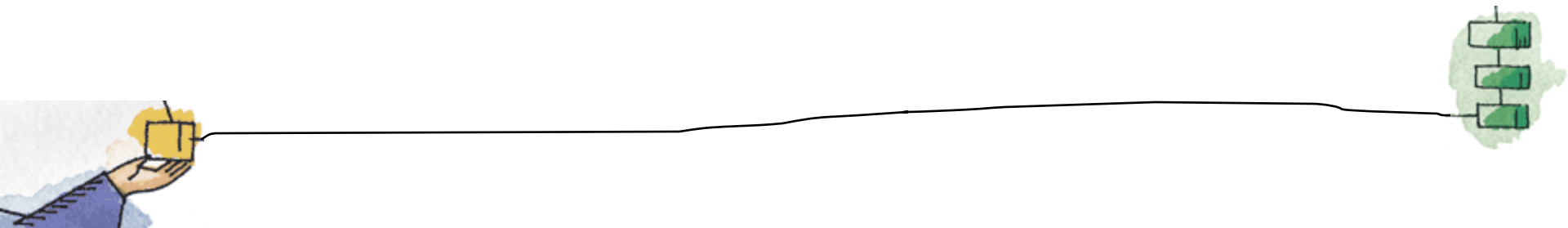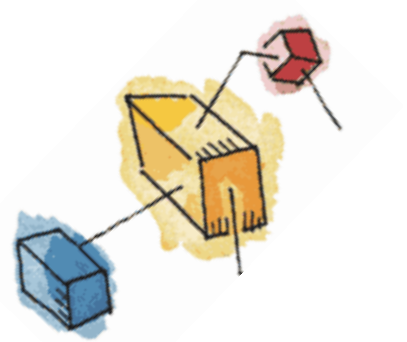
# Possible Scenario

**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.

# Solution?

- What will happen if the conditional statement is moved inside critical section?

  - Deadlock!

    - Consumer would block in critical section
    - Producer can't enter critical section to unblock consumer

# Correct Solution

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

# Using Counting 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**

# Cases to consider

- To check the working of solution, we can test the following scenarios:
  - Producer runs before consumer
  - Consumer runs before producer
  - Multiple producers run and then consumer runs

- Will the interchange of first two lines in consumer code create any issue?

# Bounded Buffer

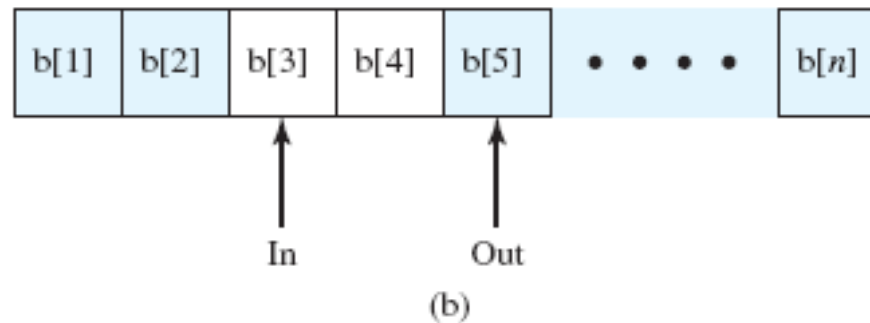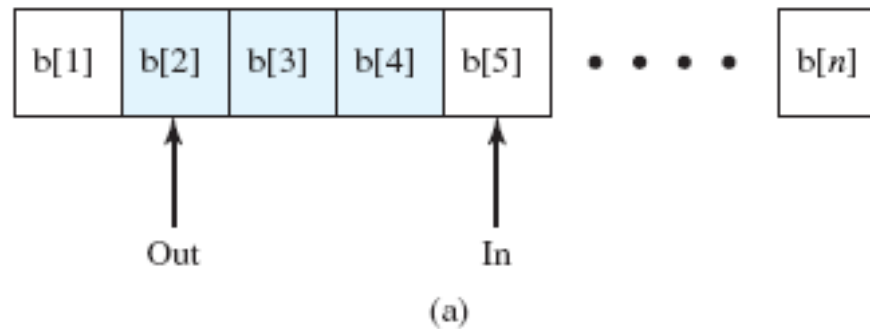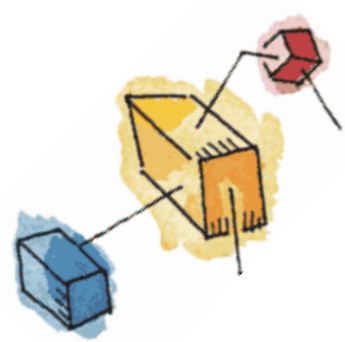| Block on: | Unblock on: |
|---|---|
| Producer: insert in full buffer | Consumer: item inserted |
| Consumer: remove from empty buffer | Producer: item removed |



Figure 5.12   Finite Circular Buffer for the Producer/Consumer Problem

# Bounded Buffer

- Three semaphores needed:

    – S: Mutual Exclusion

    – E: Keep track of empty spaces

    – N: Number of items
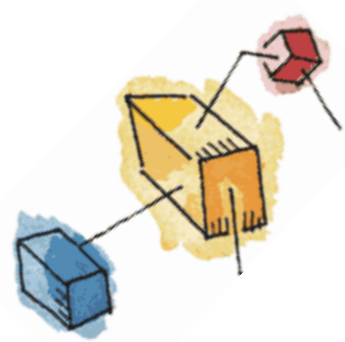
- What should be the initial values for semaphores?

# Bounded Buffer

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

# Monitors

- Semaphores are difficult to control as semWait and SemSignal calls can be scattered throughout the program.
- The Monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.

- Monitor contains:
  - One or more procedures
  - Initialization sequence
  - Local data

# Chief characteristics

1. Local data variables are accessible by monitor's procedures

2. A process enters monitor by invoking one of its procedures

3. Only one process may be executing in the monitor at a time
   – Any other process invoking monitor is blocked
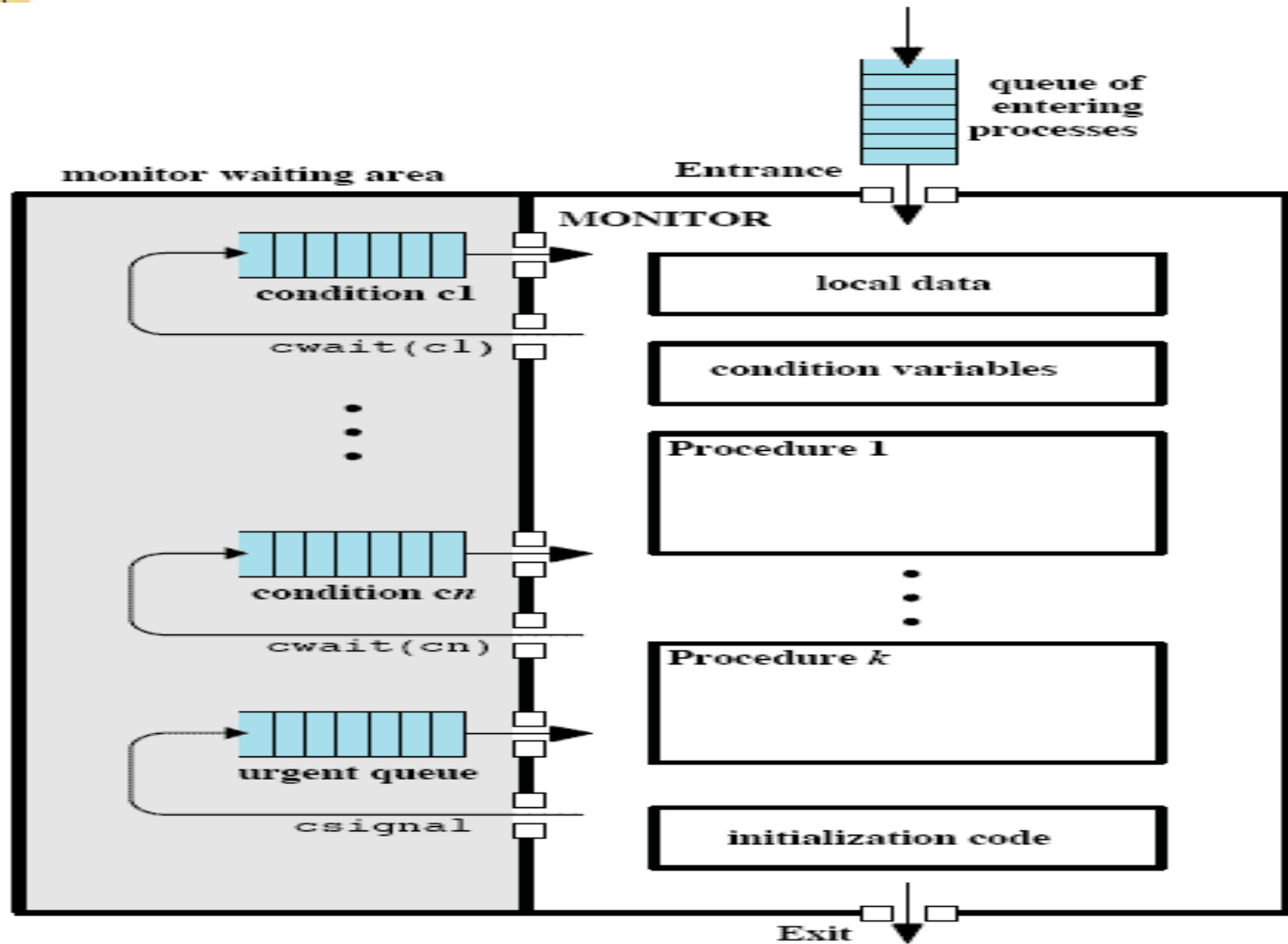
- 1st and 2nd are like OOP
- 3rd provides mutual exclusion

# Synchronization

- Synchronisation achieved by condition variables within a 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

- **Difference with Semaphores:**
  - If a process in monitor signals and no task is waiting on condition variable, the signal is lost

# Structure of a Monitor

# Structure of a Monitor

- Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time.

  - Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability.

- Once a process is in the monitor, it may temporarily block itself on condition x by issuing cwait(x)

# Structure of a Monitor

- It is then placed in a queue of processes waiting to re-enter the monitor when the condition changes, and resume execution at the point in its program following the cwait(x) call.

- If a process that is executing in the monitor detects a change in the condition variable x, it issues csignal(x), which alerts the corresponding condition queue that the condition has changed.

# 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--;                                /* one fewer item in buffer */
    csignal(notfull);                       /* resume any waiting producer */
}
{                                                   /* monitor body */
    nextin = 0; nextout = 0; count = 0;         /* buffer initially empty */
}
```
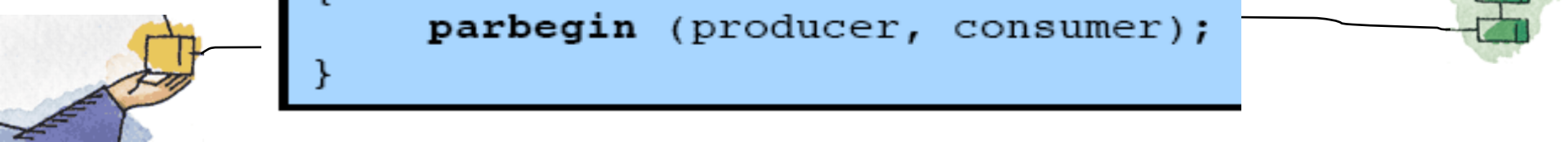
# Bounded Buffer Solution Using Monitor

- Notfull is true when there is a room to add at least once character to the buffer

- Notempty is true when there is at least one character in the buffer

- Producer can add characters to the buffer by means of append procedure only

- Consumer can remove characters from the buffer by means of take procedure only

- **"Monitor itself enforces mutual exclusion, programmer has to manage synchronization only"**

- "Semaphores require programmer to manage both"
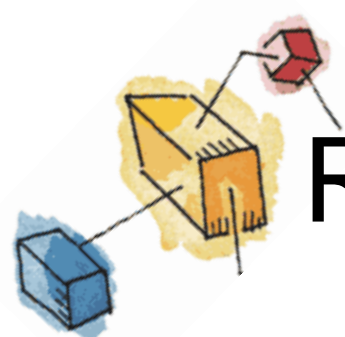
# Solution Using Monitor

```
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);
}
```

# 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.

# 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);
}
```