Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 6: Classic Approaches to Concurrent Programming
Shared variable-based synchronisation and communication
Outline

Shared variable-based synchronization and communication

- Mutual exclusion and condition synchronization
- Busy waiting
- (Suspend and resume)
- Semaphores
- Conditional critical regions
- Monitors
- Protected objects
- Synchronized methods
Two processes are **deadlocked** if each is holding a resource while waiting for a resource held by the other.

```plaintext
[Code example]
type Sem is ...;
X : Sem := 1; Y : Sem := 1;

task A;
  task body A is
  begin
    ...
    Wait(X);
    Wait(Y);
    ...
  end A;

task B;
  task body B is
  begin
    ...
    Wait(Y);
    Wait(X);
    ...
  end B;
```
Livelock

- Two processes are **livelocked** if each is executing but neither is able to make progress.

```ada
type Flag is (Up, Down);
Flag1 : Flag := Up;

task A;
task body A is
begin
  ...  
  while Flag1 = Up loop
    null;
  end loop;
  ...
end A;

task B;
task body B is
begin
  ...  
  while Flag1 = Up loop
    null;
  end loop;
  ...
end B;
```
Starvation

- an **error condition** whereby a process that wishes to gain an access to a resource, via a critical section, is **never allowed** to do so (e.g. because there are always other processes gaining access before it)
Liveness property, if a process is free of

- deadlocks
- livelocks
- starvation
A general semaphore is a non-negative integer; its value can rise to any supported positive number.

A binary semaphore only takes the value 0 and 1; the signalling of a semaphore which has the value 1 has no effect - the semaphore retains the value 1.

A general semaphore can be implemented by two binary semaphores and an integer.

With a quantity semaphore the amount to be decremented by wait (and incremented by signal) is given as a parameter; e.g. wait (S, i)
Ada does not directly support semaphores; the `wait` and `signal` procedures can, however, be constructed from the Ada synchronisation primitives.

The essence of abstract data types is that they can be used without knowledge of their implementation!
package Buffer is
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
end Buffer;

package body Buffer is
    Size : constant Natural := 32;
    type Buffer_Range is mod Size;
    Buf : array (Buffer_Range) of Integer;
    Top, Base : Buffer_Range := 0;
    Mutex : Semaphore(1);
    Item_Available : Semaphore(0);
    Space_Available : Semaphore(Size);
    procedure Append (I : Integer) is separate;
    procedure Take (I : out Integer) is separate;
end Buffer;
The Bounded Buffer

**procedure** Append(I : Integer) is begin
  Wait(Space_Available);
  Wait(Mutex);
  Buf(Top) := I;
  Top := Top+1
  Signal(Mutex);
  Signal(Item_Available);
end Append;

**procedure** Take(I : out Integer) is begin
  Wait(Item_Available);
  Wait(Mutex);
  I := BUF(base);
  Base := Base+1;
  Signal(Mutex);
  Signal(Space_Available);
end Take;
Suspended processes

- **wait** of a semaphore **delays** a process
- one method of delay is **busy waiting** (not efficient)
- other method of delay is some form of **suspension**
- when a process executes **wait** on a zero semaphore,
  - the **RTSS** (run-time support system) is **invoked**
  - the process is **removed** from the processor (CPU)
  - and placed in a queue of suspended processes
    (that is a queue of processes on that particular semaphore)
  - RTSS selects another process to run
  - **Eventually**, another process will execute a **signal** on that semaphore
  - RTSS will pick out one of the suspended processes awaiting
    a signal on that semaphore and make it executable again
Implementation of wait and signal

wait (S) :-
   if S > 0 then
      S := S - 1
   else
      number_suspended := number_suspended + 1
      suspend calling process
      S := S - 1
   end

signal (S) :-
   if number_suspended > 0 then
      number_suspended := number_suspended + 1
      make one suspended process executable again
   else
      S := S + 1
   end
Implementation of wait and signal

- **wait** and **signal** must be **indivisible (atomic)**
- with the help of RTSS possible:
  - scheduler is programmed so that it does not swap out a process while it is executing a **wait** or **signal** → non-preemptible operations
- But RTSS is not always in control of scheduling events: external actions happen asynchronously and could disturb the atomicity of the semaphore operations.
  To prohibit this:
  → RTSS will **disable interrupts** for the duration of the execution of the indivisible sequence of statements
- Disabling of interrupts is adequate for a single processor (CPU)
- Interrupts are **not** adequate for multiprocessors
Test and set

For multiprocessor: test and set instruction of processor

1. If the bit is zero then set it to one and return zero
2. If the bit is one return one

these actions are indivisible

- two parallel processes (both wishing to execute e.g. a wait will do a test and set operation on the same lock bit (initially zero)
  - one will succeed and set the bit to one
  - the other process will have returned a one will therefore have to loop round and retest the lock
  - when first process completes the wait operation it will set the bit to zero (that is unlock the semaphore)
  - other process will proceed to execute its wait operation
Criticisms of semaphores

- Semaphores are an elegant low-level synchronisation primitive, however, their use is error-prone.

- If a semaphore is omitted or misplaced, the entire program can collapse. Mutual exclusion may not be assured and deadlock may appear just when the software is dealing with a rare but critical event.

- A more structured synchronisation primitive is required.

- No high-level concurrent programming language relies entirely on semaphores.
Conditional Critical Regions (CCR)

- A critical region is a section of code that is **guaranteed** to be executed in mutual exclusion (critical section should be …)

- Shared variables are grouped together into named regions and are tagged as being resources

- Processes are **prohibited** from entering a region in which another process is already active

- Condition synchronisation is provided by guards. When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion); if the guard evaluates true it may enter, but if it is false the process is delayed

- As with semaphores, no access order can be assumed
program buffer_eg;

   type buffer_t is record
      slots : array(1..N) of character;
      size  : integer range 0..N;
      head, tail : integer range 1..N;
   end record;

buffer : buffer_t;
resource buf : buffer;

   process producer is separate;
   process consumer is separate;
end.

The Bounded Buffer
The Bounded Buffer

process producer;
  loop
    region buf when buffer.size < N do
      -- place char in buffer etc
    end region
  end loop;
end producer

process consumer;
  loop
    region buf when buffer.size > 0 do
      -- take char from buffer etc
    end region
  end loop;
end consumer
Problem

- One potential performance problem with CCRs is that processes must re-evaluate their guards every time a CCR naming that resource is left. A suspended process must become executable again in order to test the guard; if it is still false it must return to the suspended state.

- A version of CCRs has been implemented in Edison, a language intended for embedded applications, implemented on multiprocessor systems. Each processor only executes a single process so it may continually evaluate its guards if necessary.
Monitors

- A problem with CCRs is that they can be dispersed throughout the program.
- Monitors (should) alleviate this problem by providing more structured control regions.
- Monitors provide encapsulation, and efficient condition synchronisation.
- The critical regions are written as procedures and are encapsulated together into a single module.
- All variables that must be accessed under mutual exclusion are hidden; all procedure calls into the module are guaranteed to be mutually exclusive.
- Only the operations are visible outside the monitor.
- Can be found in Modula-1, Concurrent Pascal and Mesa.
The bounded buffer

monitor buffer;

export append, take;

var (*declare necessary vars*)

procedure append (I : integer);
   ...
end;

procedure take (var I : integer);
   ...
end;

begin
   (* initialisation *)
end;

How do we get condition synchronisation?
Condition variables

- Semaphores **could** be used for condition synchronisation
- But a **simpler synchronisation** primitive is introduced: **condition variables**
- In Hoare’s monitors: a condition variable is acted upon by two semaphore-like operators **wait** and **signal**
- A process issuing a **wait** is **blocked** (suspended) and placed on a queue associated with the condition variable (cf semaphores: a wait on a condition variable **always** blocks **unlike** a wait on a semaphore)
- A **blocked process** releases its hold on the monitor, allowing another process to enter
- A **signal** releases **one blocked** process. If no process is blocked then the signal has no effect (cf semaphores)
monitor buffer;

export append, take;

var BUF : array[ . . . ] of integer;
top, base : 0..size-1; NumberInBuffer : integer;

space_available, item_available : condition;

procedure append (I : integer);
begin
  if NumberInBuffer = size then
    wait (space_available);
  end if;
  BUF[top] := I;
  NumberInBuffer := NumberInBuffer+1;
  top := (top+1) mod size;
  signal (item_available)
end append;
procedure take (var I : integer);
begin
    if NumberInBuffer = 0 then
        wait (item_available);
    end if;
    I := BUF[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal (space_available);
end take;

begin (* initialisation *)
    NumberInBuffer := 0;
    top := 0; base := 0
end;

• If a process calls take when there is nothing in the buffer then it will become suspended on item_available.

• A process appending an item will, however, signal this suspended process when an item does become available.
The semantics of signal

- What happens to the signalling process and the process that is restarted? Both **must not be active** in the monitor.

- There are **various** semantics for **signal**.
The semantics of signal

- A signal is allowed only as the last action of a process before it leaves the monitor.

- A signal operation has the side-effect of executing a return statement, i.e. the process is forced to leave.

- A signal operation which unblocks another process has the effect of blocking itself; this process will only execute again when the monitor is free (proposal of Hoare).

- A signal operation which unblocks a process does not block the caller. The unblocked process must gain access to the monitor again.
POSIX Mutexes and Condition Variables

- Using **semaphores** for synchronisation and communication between threads in the same address space is **expensive** and unstructured.
- Mutex and condition variables, when combined, provide the **functionality of a monitor** but with a procedural interface.
- Mutexes and condition variables have **associated attribute objects**; we will use default attributes only.
- Example attributes:
  - set the semantics for a thread trying to lock a mutex it already has locked
  - allow sharing of mutexes and condition variables between processes
  - set/get the clock used for timeouts
POSIX Mutexes and Condition Variables

int pthread_mutex_init(pthread_mutex_t *mutex,
        const pthread_mutexattr_t *attr);
/* initialises a mutex with certain attributes */

int pthread_mutex_destroy(pthread_mutex_t *mutex);
/* destroys a mutex */
/* undefined behaviour if the mutex is locked */

int pthread_cond_init(pthread_cond_t *cond,
        const pthread_condattr_t *attr);
/* initialises a condition variable with certain attributes */

int pthread_cond_destroy(pthread_cond_t *cond);
/* destroys a condition variable */
/* undefined, if threads are waiting on the cond. variable */
POSIX Mutexes and Condition Variables

```c
int pthread_mutex_lock (pthread_mutex_t *mutex);
    /* lock the mutex; if locked already suspend calling thread */
    /* the owner of the mutex is the thread which locked it */

int pthread_mutex_trylock (pthread_mutex_t *mutex);
    /* as lock but gives an error if mutex is already locked */

int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                            const struct timespec *abstime);
    /* as lock but gives an error if mutex cannot be obtained */
    /* by the timeout */

int pthread_mutex_unlock (pthread_mutex_t *mutex);
    /* unlocks the mutex if called by the owning thread */
    /* undefined behaviour if calling thread is not the owner */
    /* undefined behaviour if the mutex is not locked */
    /* when successful, a blocked thread is released */
```
int pthread_cond_wait(pthread_cond_t *cond, 
    pthread_mutex_t *mutex);
/* called by thread which owns a locked mutex */
/* undefined behaviour if the mutex is not locked */
/* atomically blocks the caller on the cond variable and */
/* releases the lock on mutex */
/* a successful return indicates the mutex has been locked */

int pthread_cond_timedwait(pthread_cond_t *cond, 
    pthread_mutex_t *mutex, const struct timespec *abstime);
/* the same as pthread_cond_wait, except that a error is */
/* returned if the timeout expires */
POSIX Mutexes and Condition Variables

```c
int pthread_cond_signal(pthread_cond_t *cond);
/* unblocks at least one blocked thread */
/* no effect if no threads are blocked */

int pthread_cond_broadcast(pthread_cond_t *cond);
/* unblocks all blocked threads */
/* no effect if no threads are blocked */

/* all unblocked threads automatically contend for */
/* the associated mutex */
```

All functions return 0 if successful
#define BUFF_SIZE 10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    int buf[BUFF_SIZE];
} buffer;

int append(int item, buffer *B) {
    PTHREAD_MUTEX_LOCK(&B->mutex);
    while(B->count == BUFF_SIZE) {
        PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex);
    }
    /* put data in the buffer and update count and last */
    PTHREAD_MUTEX_UNLOCK(&B->mutex);
    PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
    return 0;
}
int take(int *item, buffer *B) {
    PTHREAD_MUTEX_LOCK(&B->mutex);
    while (B->count == 0) {
        PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
    }
    /* get data from the buffer and update count and first */
    PTHREAD_MUTEX_UNLOCK(&B->mutex);
    PTHREAD_COND_SIGNAL(&B->buffer_not_full);
    return 0;
}

int initialize(buffer *B) {
    /* set the attribute objects and initialize the */
    /* mutexes and condition variable */
}
Nested Monitor Calls

- What should be done if a process having made a nested monitor call is suspended in another monitor?
- The mutual exclusion in the last monitor call will be relinquished by the process, due to the semantics of the wait operation.
- However, mutual exclusion will not be relinquished by processes in monitors from which the nested calls have been made; processes that attempt to invoke procedures in these monitors will become blocked (→ performance implications).
- Maintain the lock: e.g. POSIX, Java
- Prohibit nested procedure calls altogether: e.g. Modula-1
Criticisms of Monitors

- The **monitor** gives a **structured and elegant solution to mutual exclusion** problems such as the bounded buffer.

- It does not, however, deal well with condition synchronization — requiring **low-level** condition variables.

- All the criticisms surrounding the use of semaphores apply **equally** to condition variables.
Protected Objects

- A monitor where the condition variable is replaced by guards is called a protected object.

- Combines the advantages of monitors with the advantages of conditional critical regions.

- Data and operations are encapsulated.

- Operations have automatic mutual exclusion.

- Guards can be placed on operations for condition synchronization.
**A Protected Object**

- **Encapsulates** data items and allows access to them only via protected actions — **protected subprograms** or **protected entries**

- The language **guarantees** that the **data** will only be updated **under mutual exclusion**, and that all data read will be **internally consistent**

- A protected unit may be declared as a type or as a single instance
Syntax

protected type Name (Discriminant) is

  function Fname(Params)
     return Type_Name;
  
  procedure Pname(Params);
  
  entry E1_Name(Params);

private

  entry E2_Name(Params);

  O_Name : T_Name;

end Name;
protected type Shared_Integer(Initial : Integer) is
  function Read return Integer;
  procedure Write (New_Value: Integer);
  procedure Increment (By : Integer);
private
  The_Data : Integer := Initial;
end Shared_Integer;

MyData: SharedInteger (42);
protected body Shared_Integer is

  function Read return Integer is
  begin
    return The_Data;
  end Read;

  procedure Write (New_Value : Integer) is
  begin
    The_Data := New_Value;
  end Write;

  procedure Increment (By : Integer) is
  begin
    The_Data := The_Data + By
  end Increment

end Shared_Integer;
A protected procedure provides **mutually exclusive read/write access** to the data encapsulated.

Concurrent calls to Write or Increment will be executed **one at a time**.

Protected functions provide **concurrent read only access** to the encapsulated data.

Concurrent calls to Read may be executed **simultaneously**.

Procedure and function calls are **mutually exclusive**.

The core language does not define which calls take priority.
Protected Entries and Synchronisation

- A protected entry is similar to a protected procedure in that calls are executed in mutual exclusion and have read/write access to the data.

- A protected entry can be guarded by a boolean expression (called a barrier):
  - if this barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier evaluates to true and no other tasks are currently in active inside the protected object.

- Hence protected entry calls can be used to implement condition synchronisation.
-- a bounded buffer
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get (Item : out Data_Item);
  entry Put (Item : in Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Num : Count := 0;
  Buf : Buffer;
end Bounded_Buffer;
protected body Bounded_Buffer is
  entry Get (Item : out Data_Item) when Num /= 0 is
    begin
      Item := Buf(First);
      First := First + 1;
      Num := Num - 1;
    end Get;

  entry Put (Item : in Data_Item) when Num /= Buffer_Size is
    begin
      Last := Last + 1;
      Buf(Last) := Item
      Num := Num + 1;
    end Put;

end Bounded_Buffer;

My_Buffer : Bounded_Buffer;
Barrier Evaluation

- At any instance in time, a barrier is **either open or closed**; it is **open** if the boolean expression evaluates to true, otherwise it is **closed**

- Barriers are evaluated when:
  1. a task calls one of its **protected entries** and the associated **barrier** references a variable or an attribute which **might have changed** since the barrier was last evaluated
  2. a task **executes and leaves** a protected procedure or entry, and there are tasks queued on entries whose barriers reference variables or attributes which might have changed since the barriers were last evaluated

Why are barriers not evaluated after a function call?
Write Access to a Protected Object

- task requesting read/write access
- task requesting read access
- task executing with read/write access
- task executing with read access

Chair of Software Engineering
Concurrent Object-Oriented Programming
Read Access to a Protected Object

- task requesting read/write access
- task requesting read access
- task executing with read/write access
- task executing with read access

protected object
Barrier queue
protected Resource_Control is
  entry Allocate;
  procedure Deallocate;
private
  Free : Boolean := True;
end Resource_Control;

protected body Resource_Control is
  entry Allocate when Free is
  begin
    Free := False;
  end Allocate

  procedure Deallocate is
  begin
    Free := False
  end Deallocate
end Resource_Control
The Count Attribute

- The **Count** attribute defines the number of **tasks queued on an entry**
- Its evaluation requires the read/write lock

```haskell
protected Blocker is
  entry Proceed;
private
  Release : Boolean := False;
end Blocker;

protected body Blocker is
  entry Proceed when
    Proceed'Count = 5 or
    Release is
  begin
    if Proceed'Count = 0 then
      Release := False;
    else
      Release := True;
    end if;
    end Proceed;
end Blocker;
```
protected type Broadcast is
  entry Receive(M : out Message);
  procedure Send(M : Message);
private
  New_Message : Message;
  Message_Arrived : Boolean := False;
end Broadcast;

Everyone queued on Receive should receive the Message when Send is called
protected body Broadcast is

entry Receive(M : out Message) when Message_arrived
begin
  M := New_Message
  if Receive´Count = 0 then
    Message_Arrived := False
  end if;
end Receive

procedure Send(M : Message) is
begin
  if Receive´Count > 0 then
    Message_Arrived := True
    New_Message := M;
  end if;
end Send;
end Broadcast;
Criticisms

- Ada does **not fully** integrate its models of **concurrent** and **object-oriented** programming

- Neither **tasks** nor **protected objects** are **extensible**
Synchronized Methods

- Java provides a mechanism by which monitors can be implemented in the context of classes and objects.
- In Java, there is a lock associated with each object which cannot be accessed directly by the application but is affected by:
  - the method modifier synchronized
  - block synchronization.
- When a method is labeled with the synchronized modifier, access to the method can only proceed once the lock associated with the object has been obtained.
- Hence synchronized methods have mutually exclusive access to the data encapsulated by the object, if that data is only accessed by other synchronized methods.
- Non-synchronized methods do not require the lock and, therefore, can be called at any time.
Example of Synchronized Methods

```java
public class SharedInteger {
    private int theData;

    public SharedInteger(int initialValue) {
        theData = initialValue;
    }

    public synchronized int read() {
        return theData;
    }

    public synchronized void write(int newValue) {
        theData = newValue;
    }

    public synchronized void incrementBy(int by) {
        theData = theData + by;
    }
}

SharedInteger myData = new SharedInteger(42);
```

Chair of Software Engineering
Concurrent Object-Oriented Programming
Block Synchronization

- Provides a mechanism whereby a block can be labeled as synchronized
- The synchronized keyword takes as a parameter an object whose lock it needs to obtain before it can continue
- Hence synchronized methods are effectively implementable as

```java
public int read() {
    synchronized (this) {
        return theData;
    }
}
```
- Where this is the Java mechanism for obtaining the current object
Warning

- Used in its full generality, the **synchronized block** can **undermine** one of the **advantages** of monitor-like mechanisms, that of **encapsulating synchronization constraints** associate with an object into a single place in the program.

- This is because it is **not possible to understand** the synchronization associated with a particular object by just looking at the object itself when other objects can name that object in a synchronized statement.

- However with **careful use**, this facility **augments** the basic model and allows **more expressive synchronization constraints** to be programmed.
Static Data

- Static data is shared between all objects created from the class.
- To obtain mutually exclusive access to this data requires all objects to be locked.
- In Java, classes themselves are also objects and therefore there is a lock associated with the class.
- This lock may be accessed by either labeling a static method with the synchronized modifier or by identifying the class's object in a synchronized block statement.
- The latter can be obtained from the Object class associated with the object.
- Note, however, that this class-wide lock is not obtained when synchronizing on the object.
class StaticSharedVariable {
    private static int shared;
    ...

    public int Read() {
        synchronized (this.getClass()) {
            return shared;
        }
    }

    public void Write(int I) {
        synchronized (this.getClass()) {
            shared = I;
        }
    }
}

Could have used:
public static synchronized void Write(int I)
Waiting and Notifying

- To obtain **conditional synchronization** requires the methods provided in the predefined object class:
  ```java
  public void wait() throws InterruptedException;
  // also throws IllegalMonitorStateException
  public void notify();
  // throws IllegalMonitorStateException
  public void notifyAll();
  // throws IllegalMonitorStateException
  ```

- These methods should be used **only** from within methods which **hold the object lock**

- If called **without the lock**, the exception `IllegalMonitorStateException` is **thrown**
Waiting and Notifying

- The `wait` method **always blocks the calling thread and releases the lock associated with the object**
- A wait **within a nested monitor releases only the inner lock**
- The `notify` method wakes up **one waiting thread**; the one woken is **not defined** by the Java language
- `notify` **does not release the lock**; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up **all** waiting threads requires use of the `notifyAll` method
- If no thread is waiting, then `notify` and `notifyAll` **have no effect**
Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread.

- In this case the InterruptedException is thrown.
There are no explicit condition variables. An awoken thread should usually evaluate the condition on which it is waiting (if more than one exists and they are not mutually exclusive)

```java
public class BoundedBuffer {
    private int buffer[];
    private int first;
    private int last;
    private int numberInBuffer = 0;
    private int size;

    public BoundedBuffer(int length) {
        size = length;
        buffer = new int[size];
        last = 0;
        first = 0;
    }
}
```
public synchronized void put(int item) throws InterruptedException {
    if (numberInBuffer == size) {
        wait();
    }
    last = (last + 1) % size ; // % is modulus
    numberInBuffer++;
    buffer[last] = item;
    notify();
}

public synchronized int get() throws InterruptedException {
    if (numberInBuffer == 0) {
        wait();
    }
    first = (first + 1) % size ; // % is modulus
    numberInBuffer--;
    notify();
    return buffer[first];
}
public synchronized void put(int item) throws InterruptedException {
    while (numberInBuffer == size) {
        wait();
    }
    last = (last + 1) % size; // % is modulus
    numberInBuffer++;
    buffer[last] = item;
    notify();
}

public synchronized int get() throws InterruptedException {
    while (numberInBuffer == 0) {
        wait();
    }
    first = (first + 1) % size; // % is modulus
    numberInBuffer--;
    notifyAll();
    return buffer[first];
}

Mutually exclusive waiting
Inheritance and synchronization

- Combination of object-orientation paradigm with mechanism for concurrent programming may lead to so-called **inheritance-anomaly**

- Inheritance-anomaly **exists** if
  - the synchronisation between operations of a class is **not local**, but may depend on the whole set of operations present for the class
  - e.g. when a subclass adds new operations, it may be necessary to change the synchronisation defined in the parent (ancestor) class
public class BoundedBuffer {
    private int buffer[];
    private int first;
    private int last;
    private int numberInBuffer = 0;
    private int size;

    public BoundedBuffer(int length) {
        size = length;
        buffer = new int[size];
        last = 0;
        first = 0;
    }
}
Bounded Buffer

```java
public synchronized void put(int item) throws InterruptedException {
    if (numberInBuffer == size) {
        wait();
    }
    last = (last + 1) % size; // % is modulus
    numberInBuffer++;
    buffer[last] = item;
    notify();
}

public synchronized int get() throws InterruptedException {
    if (numberInBuffer == 0) {
        wait();
    }
    first = (first + 1) % size; // % is modulus
    numberInBuffer--;
    notify();
    return buffer[first];
}
```
public class AccessError extends Exception {
}

public class ControlledBoundedBuffer extends BoundedBuffer {
    // incorrect code

    boolean prohibited;

    ControlledBoundedBuffer (int length) {
        super (length);
        prohibited = false,
    }

    public synchronized void prohibitAccess() throws InterruptedException {
        if (prohibited) wait();
        prohibited = true;
    }

    public synchronized void allowAccess() throws AccessError {
        if (!prohibited) throw new AccessError();
        prohibited = false;
        notify();
    }
}
Summary

- **critical section** — code that must be executed under mutual exclusion
- **producer-consumer system** — two or more processes exchanging data via a finite buffer
- **busy waiting** — a process continually checking a condition to see if it is now able to proceed
- **livelock** — an error condition in which one or more processes are prohibited from progressing whilst using up processing cycles
- **deadlock** — a collection of suspended processes that cannot proceed
- **starvation (indefinite postponement)** — a process being unable to proceed as resources are not made available
Summary

- **semaphore** — a non-negative integer that can only be acted upon by `wait` and `signal` atomic procedures
- two more structured primitives are: **condition critical regions and monitors**
- suspension in a monitor is achieved using **condition variable**
- **POSIX mutexes and condition variables** give monitors with a procedural interface
- Ada’s **protected objects** give structured mutual exclusion and high-level synchronization via barriers
- Java’s **synchronized methods** provide monitors within an object-oriented framework