Concurrency

- **Motivation 1: Mutual Exclusion**
  - **John and Mary share a bank acc't**
  - **withdraw x =**

    copy balance to local machine

    subtract x

    give out $$

    write back (balance - x)
  - **Suppose John & Mary each withdraw $100 at the same time, from different machines:**

    John copies balance

    Mary copies balance

    John gets $100

    Mary gets $100

    John writes back (balance - 100)

    Mary writes back (balance - 100)
  - **New balance = balance - 100!**

Mutual Exclusion (continued)

- **Use a variable to restrict access to the account:**

  type gate = {open, closed};

  var access: gate;

  - **John/Mary:**

    while access = open do;

        access := closed;

        withdraw $$;

        access := open

  - **But what if**

    John : test, access = open

    Mary : test, access = open -- before John has closed it

    John : set access = closed

    Mary : set access = closed

    John: withdraw $$

    Mary: withdraw $$

    etc.

- **Problem: test/set of access is *divisible***

The Producer/Consumer Model

- **Motivation 2: Synchronization**
  - **Producer / Consumer:**

    Producer makes items, places them in n-element buffer

    Consumer removes items from buffer
  - **Important:**

    don't put items in full buffer

    don't take items from empty buffer
  - **Suppose a buffer counter t is incremented by producer and decremented by consumer:**

    read t into private register

    update value of t locally

    write back to t

Semaphores (Dijkstra 1965)

- **A *semaphore* is a data object that can assume an integer value and can be operated on by primitives P and V.**
  - **P(s) =**

    if s > 0 then s := s - 1

    else suspend current process;
  - **V(s) =**

    if there is a process suspended on s

    then wake it up

    else s := s + 1;
  - **Important:**

    P and V are *indivisible* instructions.

P: *proberen* (to try) or *passeren* (to pass)
V: *verhogen* (to increase) or *vrygeren* (to release)

Solving Bank Problem with Semaphores

var mutex : semaphore := 1;

- **John:**

  P(mutex);

  withdraw $$;

  V(mutex);

- **Mary:**

  P(mutex);

  withdraw $$;

  V(mutex);

General Producer/Consumer Model

```
semaphore   mutex := 1,        -- availability control
            in := 0,           -- # of things in buffer
            spaces := n;       -- # of empty spaces in buffer

process producer
    repeat
        produce thing;
        P(spaces);             -- wait for buffer space
        P(mutex);              -- wait for buffer availability
        put thing in buffer;
        V(mutex);              -- free buffer
        V(in);                 -- increase # of items in buffer
    forever

process consumer
    repeat
        P(in);                 -- wait until something in buffer
        P(mutex);              -- wait for buffer availability
        take thing from buffer;
        V(mutex);              -- free buffer
        V(spaces);             -- increase # free spaces in buffer
    forever
```

About Semaphores

- **Each semaphore has**
  - **way to suspend processes (use process queue)**
  - **policy for selecting process to wake up.**
- **One semaphore per synchronization condition, not per resource.**
- **Low level, may be tricky and tedious to use.**
- **Deadlock quite possible.**

Monitors (Brinch Hansen & Hoare '73-'74)

- **ADTs in a concurrent environment**
  - **used in Concurrent Pascal, Modula**

- **Instance of a monitor => shared resource**

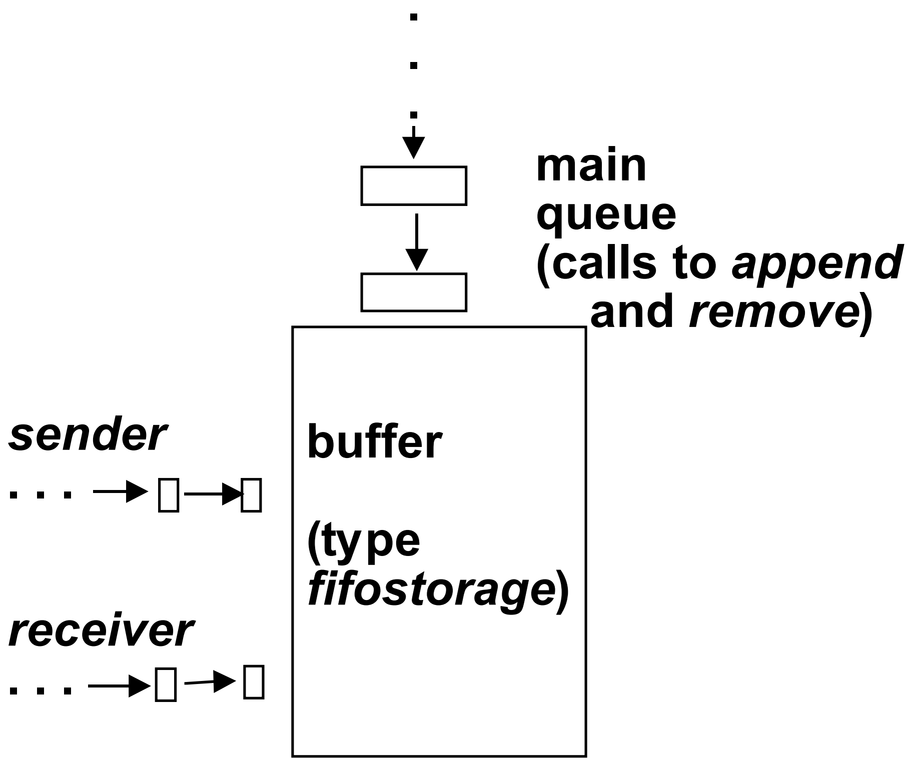
- **Monitors are passive:**

  data + proc defs + init code

- **Active processes <u>use</u> monitors**

- **Mutual exclusion of access to monitor guaranteed by system**

Implementing a Producer/Consumer Buffer with Monitors



Monitor Implementation

```
type fifostorage =
monitor
        var   contents: array [1. .n] of integer;               -- data
              tot: 0. .n;                -- count of items in buffer
              in, out: 1. .n;            -- "pointers" to buffer cells
              sender, receiver: queue;

        procedure entry append (item: integer);                 -- procedures
        begin
            if tot = n then delay (sender);
            contents[in] := item;
            in := (in mod n) + 1;
            tot := tot + 1;
            continue (receiver)
        end;

        procedure entry remove (var item: integer);
        begin
            if tot = 0 then delay (receiver);
            item := contents[out];
            out := (out mod n) + 1;
            tot := tot - 1;
            continue (sender)
        end;

begin
    --  initialization code
        tot := 0;
        in := 1;
        out := 1
end
```

Monitor Implementation (continued)

```
type producer = process (storage: fifostorage);
var    element: integer;
begin cycle
    . . .
    storage.append (element);
    end
end;

type consumer = process (storage: fifostorage);
var    datum: integer;
begin cycle
    . . .
    storage.remove (datum);
    . . .
    end
end;

var    meproducer: producer;
       youconsumer: consumer;
       buffer: fifostorage;

begin -- start everything
    init buffer, meproducer (buffer), youconsumer (buffer)
end
```

Monitors (continued)

- **for cooperation, use *delay* and *continue*:**
  - **delay -- takes name of queue and suspends executing process on that queue**
  - **continue -- takes name of queue and reactivates a suspended process on that queue.**
- **In both cases, active process releases lock on monitor.**

Rendezvous (Ada)

- **Ada concurrent units: *tasks***
  - **No active/passive distinction; shared resource is represented by a task.**
  - **Entry into task is via an *accept* statement, often inside a *select*, i.e.,**

    {when <condition> =>}

        accept <entryname> (<params>) do <entry body>;

    end;
  - **To other process, task entry call is (and looks) just like any procedure except it's only carried out when the task owning the entry executes the corresponding *accept*.**
  - **Rendezvous:**

    entry has been invoked, **and**

    task w/entry declaration has executed accept.
  - **Suspension:**

    caller invokes entry when task not in accept, **or**

    task executes accept when no other task has called entry

Ada Rendezvous (continued)

- **Accepts:**
  - **Alternatives with true *when* condition are marked open. (Those without conditions are always open.)**
  - **Open entries for which an entry call has already been issued are marked available. Any available alternative may be selected (nondeterminism).**
  - **Open alternatives but no available alternatives => task suspends until one becomes available.**
  - **No open alternatives => error.**
  - **Only one entry can be accepted at a time**

Either-Or Rendezvous Task

```
task body  Data_collector is
begin
    select
        -- if data is available for processing, process it
        -- otherwise execute the else part of the select statement
        accept Put_data ( Sensor_in: SENSOR_VALUE) do
                Process_data (Sensor_in) ;
        end Put_data ;
    else
        -- execute Self_test rather than wait for data
        Self_test ;
    end select ;
end  Data_collector ;
```

Implementing Producer/Consumer with Ada Rendezvous

```
task buffer_handler is
    entry append (item: in integer);
    entry remove (item: out integer);
end;
task body buffer_handler is
    n: constant integer := 20;                -- buffer size
    contents: array (1. .n) of integer;
    in, out: integer range 1. .n := 1;    -- "pointers" into buffer
    tot: integer range 0. .n := 0;            -- # of items currently in buffer
begin loop
    select
        when tot < n =>                    -- buffer not full
            accept append (item: in integer) do
                contents(in) := item;
            end;
        in := (in mod n) + 1;
        tot := tot + 1;
    or
        when tot > 0 =>                    -- buffer not empty
            accept remove (item: out integer) do
                item := contents (out);
            end;
        out := (out mod n) + 1;
        tot := tot - 1;
    end select;
end loop;
end buffer_handler;
```

```
PRODUCER                              CONSUMER
loop                                  loop
  produce new value V;                    buffer_handler.remove (V);
  buffer_handler.append (V);              consume V;
  exit when V => end of stream;           exit when V => end of stream;
end loop;                             end loop;
```

Ada Sequence Counter

```
task Counter is
        entry Add (N: NATURAL) ;
        entry Subtract (N: NATURAL) ;
        entry Put_value (N: NATURAL) ;
        entry Get_value (N: out NATURAL) ;
end Counter ;

task body  Counter is
        Value: NATURAL := 0 ;
begin
        loop
                select
                        accept Add (N: NATURAL) do
                                Value := Value + N ;
                        end Add ;
                or
                         accept Subtract (N: NATURAL) do
                                Value := Value - N ;
                        end Subtract ;
                   or
                        accept Put_value (N: NATURAL) do
                                Value := N ;
                        end Put_value ;
                   or
                        accept Get_value (N: out NATURAL) do
                                N := Value ;
                        end Get_value ;
                end select ;
        end loop ;
end Counter ;
```

Ada Transponder

```
task type Transponder is
    entry  Give_position (Pos: POSITION ) ;
end Transponder ;

task body Transponder is
        Current_position: POSITION ;
        C1, C2: Satellite.COORDS ;
        loop
        select
                accept Give_position (Pos: out POSITION) do
                        Pos:= Current_position ;
                end Give_position ;
        else
                C1 := Satellite1.Position ;
                C2 := Satellite2.Position ;
                Current_position := Navigator.Compute (C1, C2) ;
        end select ;
        end loop ;
end Transponder ;
```

Concurrent Office IR System

```
procedure Office_system is
        task Get_command  ;
        task Process_command is
                entry Command_menu ;
                entry Display_indexes ;
                entry Edit_qualifier ;
                 -- Additional entries here.
                 -- One for each command
        end Process_commands ;
        task Output_message is
        entry  Message_available ;
        end Output_message ;
         task Workspace_editor is
        entry Enter ;
        entry Leave ;
        end Workspace_editor ;
```

(to be continued)

Concurrent Office IR System - II

```
task body  Get_command is
begin
        loop
                Cursor_position := Get_cursor_position ;
                exit when cursor positioned in workspace or
        (cursor positioned over menu and button pressed)
                Display_cursor_position ;
        end loop ;
        if In_workspace (Cursor_position) then
                Workspace_editor.Enter ;
        elsif In_command_menu  (Cursor_position) then
                Process_command.Command_menu ;
        elsif In_Known_indexes  (Cursor_position) then
                Process_command.Display_indexes ;
        elsif In_Current_indexes  (Cursor_position) then
                                                ...
                                                Other commands here
                                        ...
end Get_command ;
```

Concurrent Office IR System - III

```
task body Process_command is
        Command: COMMAND.T ;
        Index: INDEX.T ;
begin
        Workspace_editor.Leave ;
        loop
                accept Command_menu do
                Display_command_menu ;
                Get_menu_selection (Command) ;
                Execute_menu_command (Command) ;
        end Command_menu;
        accept Display_indexes do
                Display_current_indexes ;
                Get_index_selection (Index) ;
        end Display_indexes;
        ...
    Other commands here
    ...
end Office_system ;
```

Sequenced Rendezvous Actions

- **For the case where actions are to be in a strict sequence**

```
task body Thermocouple is
begin
        accept Get_temperature (T: in out TEMPERATURE) do
                -- code here to interrogate the hardware
        end Get_temperature ;
        accept  Calibrate (T: TEMPERATURE)  do
                 -- code here to calibrate the thermocouple
        end Calibrate ;
        accept Disconnect  do
                -- code to implement a hardware shutdown
        end Disconnect ;
end Thermocouple ;
```