# Synchronization of Concurrent Processes

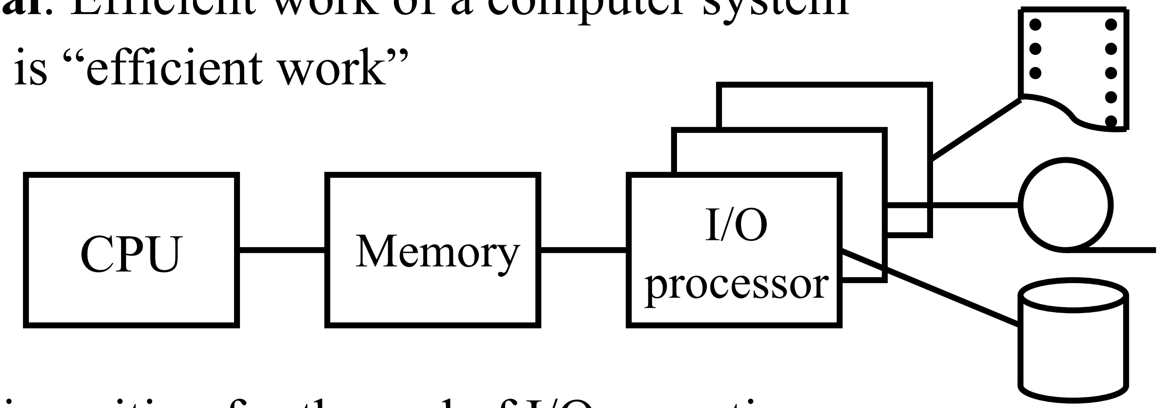
Trifon Ruskov
ruskov@tu-varna.acad.bg

Technical University of Varna - Bulgaria

# Work of a computer system

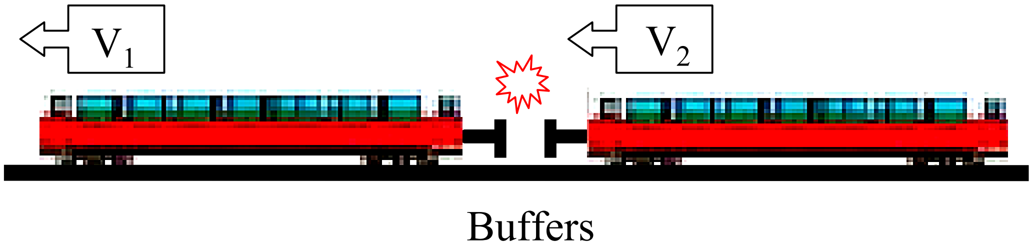
**Main goal**: Efficient work of a computer system

- What is “efficient work”



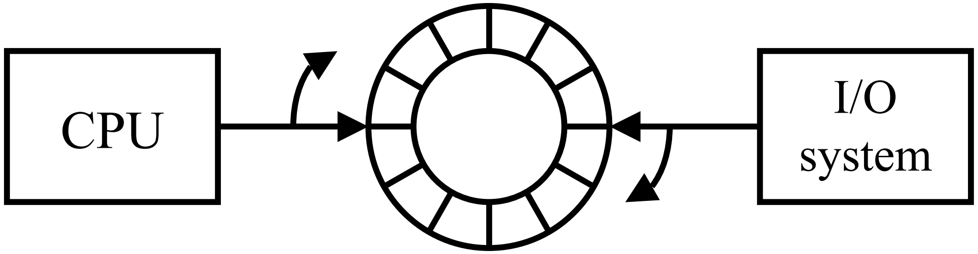
- CPU is waiting for the end of I/O operations

## Two asynchronous moving systems



- If $V_1 \neq V_2$ , then crash
- If the system is “a computer system”, then non efficient work $\equiv$ waiting

# Round buffer



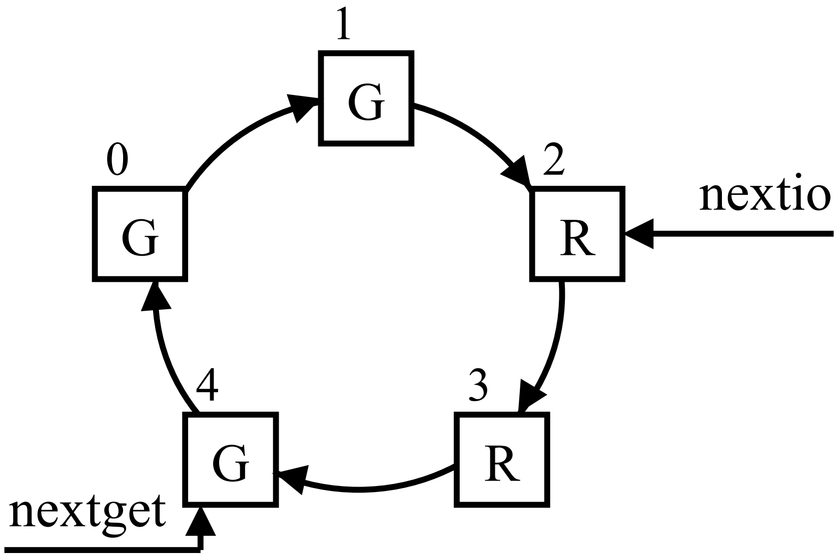
array: Buf[0], Buf[1], … , Buf[n-1]

After Buf[I] follows Buf[(I+1) mod n]

Typical computation process (program):

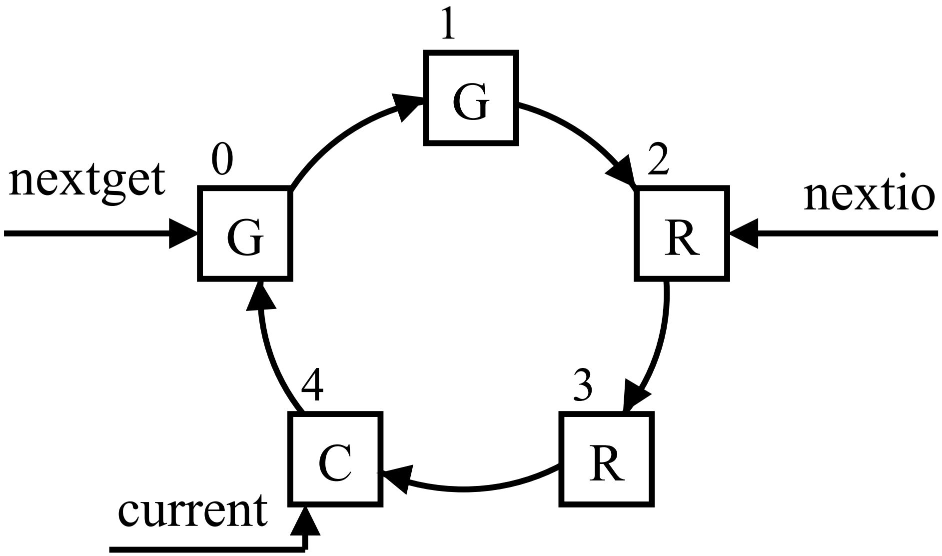
```
...
GetBuf;                               Receive a full buffer
Compute(Buf[current]);
ReleaseBuf;                           Free the buffer
...
```

# Round buffer (cont.)



in *GetBuf* procedure:

```
current := nextget;
nextget := (nextget+1)mod5;
```



G – full buffer (advance buffer)
R – empty buffer

# Round buffer (cont.)

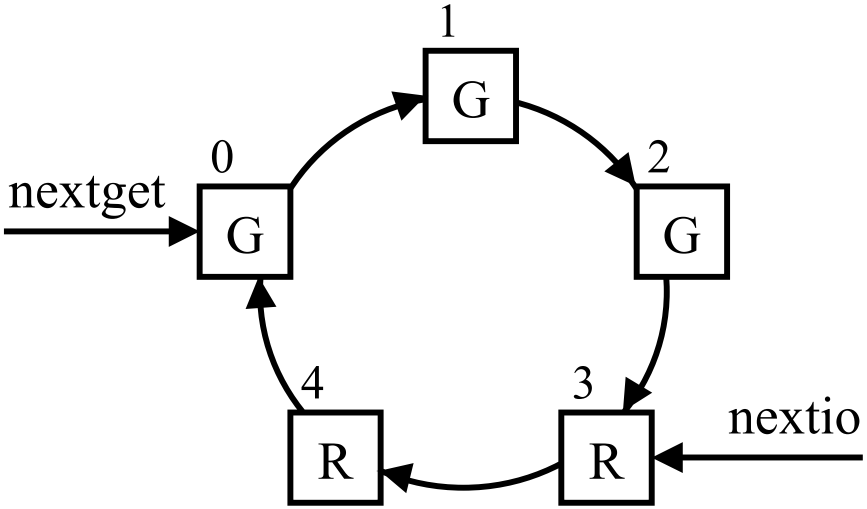
Operation *Read(Buf[nextio])* is asynchronous

after *Read* …

```
nextio := (nextio + 1) mod 5;
```

*ReleaseBuf* procedure:

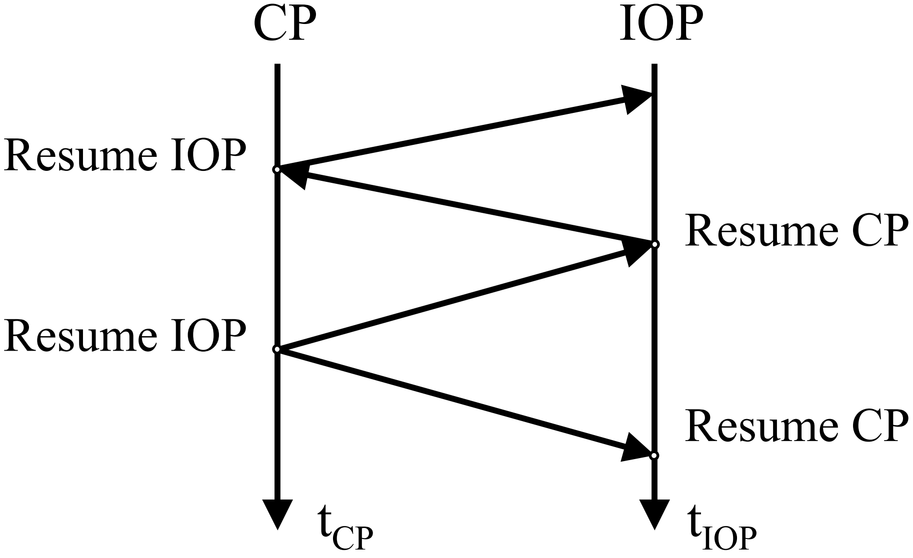
*Buff[current]* is marked as free (empty) - R



# Round Buffer Implementation

CP – Computational program
IOP – Input/Output program

## Co-routines



Resume ≡ continue execution of …

# Round Buffer Implementation (cont.)

*n* – total number of buffers
*r* – R-type buffers (empty)
*ch* – channel (I/O processor)

**in CP:**

```
procedure GetBuf;
begin
  repeat
    if not busy[ch] then resumeIOP;
  until not (r = n);          Continue only if there is a full buffer
  current := nextget;
  nextget := (nextget + 1) mod n;
end;

procedure ReleaseBuf;
begin
  r := r + 1;
  if not busy[ch] then resumeIOP;
end;
```

# Round Buffer Implementation (cont.)

**in IOP:**

```
repeat
  while r = 0 then resumeCP;
  Read(ch, Buf[nextio]);
  resumeCP;
  nextio := (nextio + 1) mod n;
  r := r - 1;
until forever;
```

**Initialization:**

```
nextio := 0;
nextget := 0;
```

- ✓ What is the value of *n* ?

# The Problem: I/O System is Waiting

**Everywhere in CP program:**

```
if not busy[ch] then resumeIOP;
```

- ✓ But how often ?

**Solution:**

Signal from I/O system≡ interrupt

- ✓ What is the meaning of the interrupt from an I/O device (system) ?

# I/O Channel Interrupts CPU

```
procedure GetBuf;
begin
  while (r = n) do ;      All buffers are empty
  current := nextget;
  nextget := (nextget + 1) mod n;
end;

procedure ReleaseBuf;
begin
  r := r + 1;
  if not busy[ch] then Read(ch, Buf[nextio]);
end;
```

# Interrupt Procedure

```
procedure IR;
begin
  SaveCurrentState;
  nextio := (nextio + 1) mod n;
  r := r - 1;
  if r <> 0 then Read(ch, Buf[nextio]);
  RestoreState;
end;

procedure Init;
begin
  r := n;
  nextget := 0;
  nextio := 0;
  Read(ch, Buf[nextio]);
end;
```

# The Problem:

- ✓ When is the interrupt accepted ?

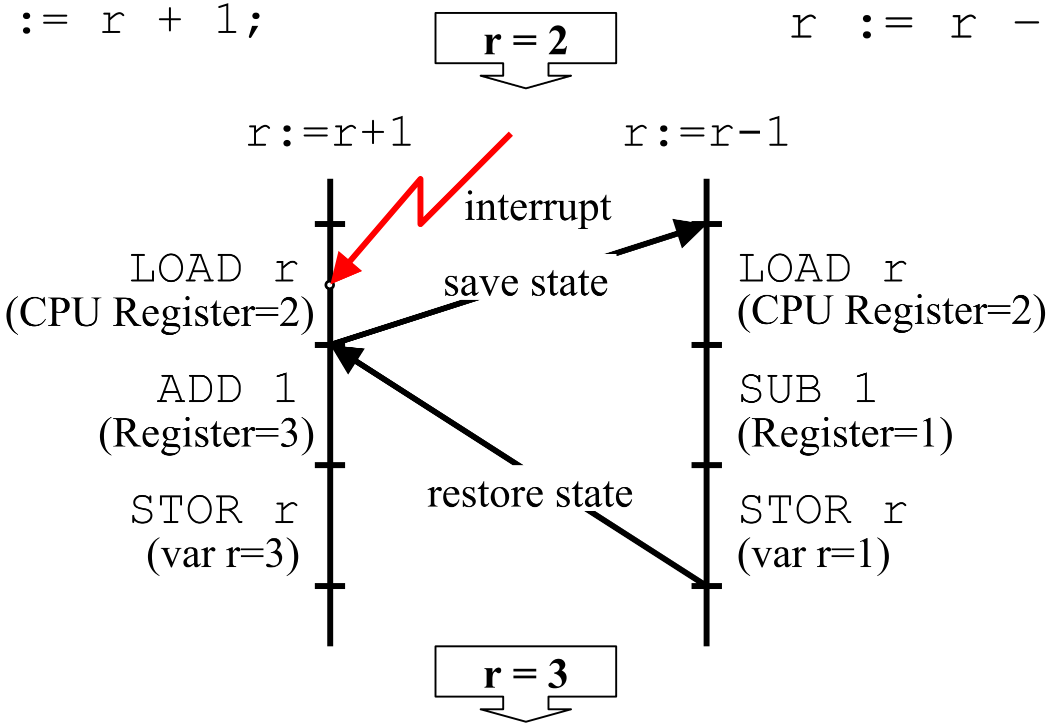
*A closer look at the previous program*

in ReleaseBuf:

```
r := r + 1;
```

in IR:

```
r := r - 1;
```



**r = 2**

r:=r+1

r:=r-1

interrupt

save state

restore state

LOAD r
(CPU Register=2)

ADD 1
(Register=3)

STOR r
(var r=3)

LOAD r
(CPU Register=2)

SUB 1
(Register=1)

STOR r
(var r=1)

**r = 3**

✓ But *r* must be 2 !

# Processes

- **Informal definition:**

  A sequential process is the activity, resulting from the execution of a program with its data by a sequential processor (CPU).

- **Conceptually:**

  Each process has its own processor and program stored in physical memory.

- **In reality:**

  Two different processes may share the same processor or the same program.

- **Therefore:**

  A process is not equivalent to a program and is not equivalent to a processor (CPU) !

# Processes (cont.)

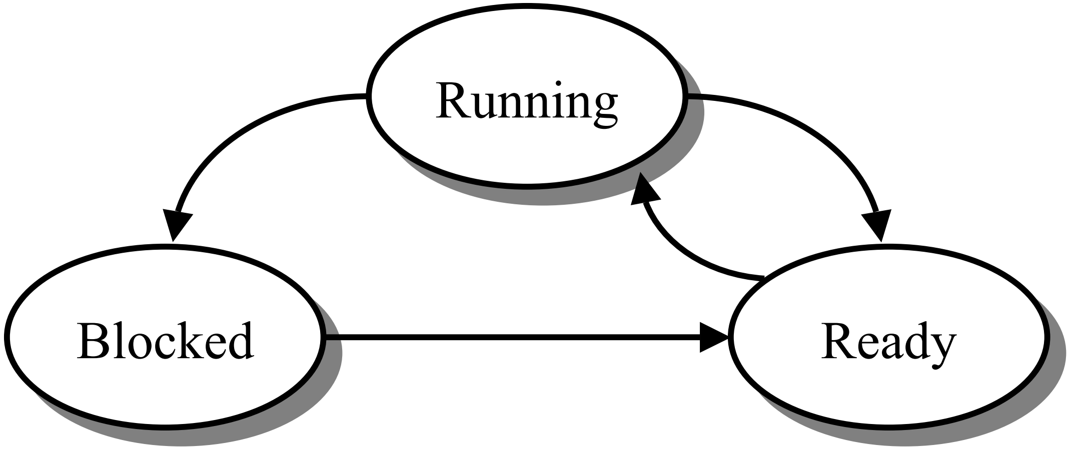
Running every process is described by a sequence of vectors $S_0$, $S_1$, … $S_i$, … , and every vector contains at least the program counter and CPU registers.

The kernel creates the illusion of a separate CPU for each running process. The kernel may also provide separate storage (virtual memory) for each process.

**More formal definition:**

A process is ordered triple <*CPU, program, data*> in execution.

# Process State Diagram



- ✓ What is the number of processes in every state? Max number? Min number?
- ✓ Null process for easier scheduling implementation.

# Critical Section (CS)

This part of a process program in which access to common resources (common data in particular) is made.

**Assumptions about the system:**

1. Writing into and reading from the common memory are both indivisible operations.
2. Critical sections may not have priorities associated with them.
3. The relative speeds of the processes are unknown.
4. A program may halt only outside its CS.

# Software Solution (Dijkstra, 1968)

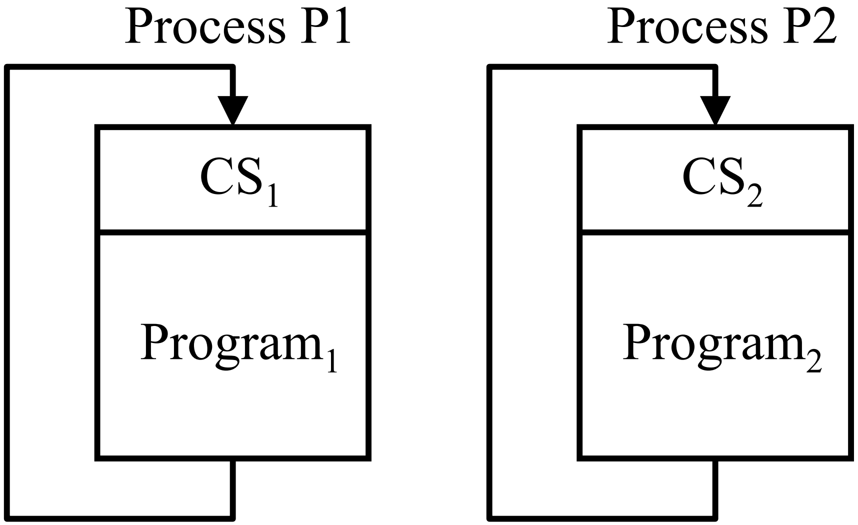
**Our aim:**

Prevent P1 and P2 from entering their CSs at the same time (mutual exclusion)

**Three possible types of blocking must be avoided:**

1. A process running outside its CS can not prevent another process from entering its CS.
2. It must not be possible for one of the processes to repeatedly enter its CS while the other process never gets a chance.
3. The processes about to enter their CSs can not, by entering infinite waiting loops.

# Software Solution (Dijkstra, 1968) (cont.)



```
parbegin
  P1: repeat
         CS1;
         program1;
      until forever;
  P2: repeat
         CS2;
         program2;
      until forever;
parend;
```

# Incorrect Solution

```
I. var turn: integer := 2;
    parbegin
      P1: repeat
            while turn = 2 do ;      { wait loop }
            CS1;
            turn := 2;
            program1;
          until forever;
      P2: repeat
            while turn = 1 do ;      { wait loop }
            CS2;
            turn := 1;
            program2;
          until forever;
    parend;
```

✓ Violating requirement 1

# Incorrect Solution (cont.)

```
II.var C1, C2: boolean := true;
   parbegin
     P1: repeat
          A1:  C1 := false;
          B1:  while not C1 do ;
               CS1;
               C1 := false;
               program1;
         until forever;
     P2: { analogous to P1 }

   parend;
```

- ✓ Mutual blocking

# Incorrect Solution (cont.)

```
III.var C1, C2: boolean := true;
    parbegin
     P1: repeat
            C1 := false;
            if not C2 then C1 := true;
              else begin
                 CS1;
                 C1 := true;
                 program1;
              end;
         until forever;
     P2: { analogous to P1 }

   parend;
```

- ✓ 2nd and 3rd type of blocking

# The First Complete Solution of the Critical Region Problem (T.Dekker, 1966)

```
var C1, C2: boolean := true;
    turn: integer := 1;
parbegin
    P1: repeat
          C1 := false;
          while not C2 do
              if turn = 2 then
                begin
                  C1 := true;
                  while turn = 2 do ;
                  C1 := false;
                end;
          CS1;
          turn := 2;
          C1 := true;
          program1;
        until false;
    P2:  . . .
parend;
```

# Peterson (1981). A Simple and Elegant Algorithm

```
var C1, C2: boolean := true;
    turn: integer;
parbegin
    P1: repeat
            C1 := false;
            turn := 1;
            while not C2 and turn = 2 do ;
            CS1;
            C1 := true;
            program1;
        until false;
    P2: . . .

parend;
```

# Why do we need another solution ?

**Problems with the Dekker & Peterson algorithms:**

1. The solutions are too complex and hard for more that 2 processes.

2. During the time when one process is in its CS, another is consuming CPU time.

# Semaphores. (Dijkstra, 1968)

**Semaphore** - a nonnegative integer variable *s* on which only two operations are defined - *P* and *V*.

1. *P(s):* tries to execute $s := s - 1$
   if possible then the process continues
   if not possible $(s = 0)$, the process waits until $s > 0$

2. *V(s):* executes $s := s + 1$
   if there is a process waiting to complete its *P(s)* operation, it wakes up and continues execution

The *P(s)* and *V(s)* operations are indivisible.

# Mutual Exclusion. A Solution for N Processes

```
var mutex: semaphore := 1;
parbegin
  P1: repeat ... until forever;
        ...
  Pi: repeat
        P(mutex);
        CSi;
        V(mutex);
        program_i;
      until forever;
      ...
  Pn: repeat ... until forever;
parend;
```

- General semaphores
- Binary semaphores

# Producer-Consumer Problem

```
var empty: semaphore := n;   { number of empty buffers}
    full: semaphore := 0;    { number of full buffers }
    me: semaphore := 1;      { mutual exclusion }
parbegin
  producer: repeat
                produce_data;
                P(empty);
                P(me);
                add_to_buffer;
                V(me);
                V(full)
            until forever;
  consumer: repeat
                P(full);
                P(me);
                take_from_buffer;
                P(me);
                P(empty);
                process_data;
            until forever;
parend;
```

# Implementation of Semaphore Operations

**A problem:**

It is hard to provide directly hardware implementations of *P* and *V* as CPU instructions.

**TS(x) instruction**

```
function TS(x: boolean): boolean;
  begin
    TS := x;
    x := false;
  end;
```

```
P(s):     while TS(s) do ;
V(s):     s := true;
```

- ✓ A problem: “Busy wait”

# Avoiding the Busy Wait

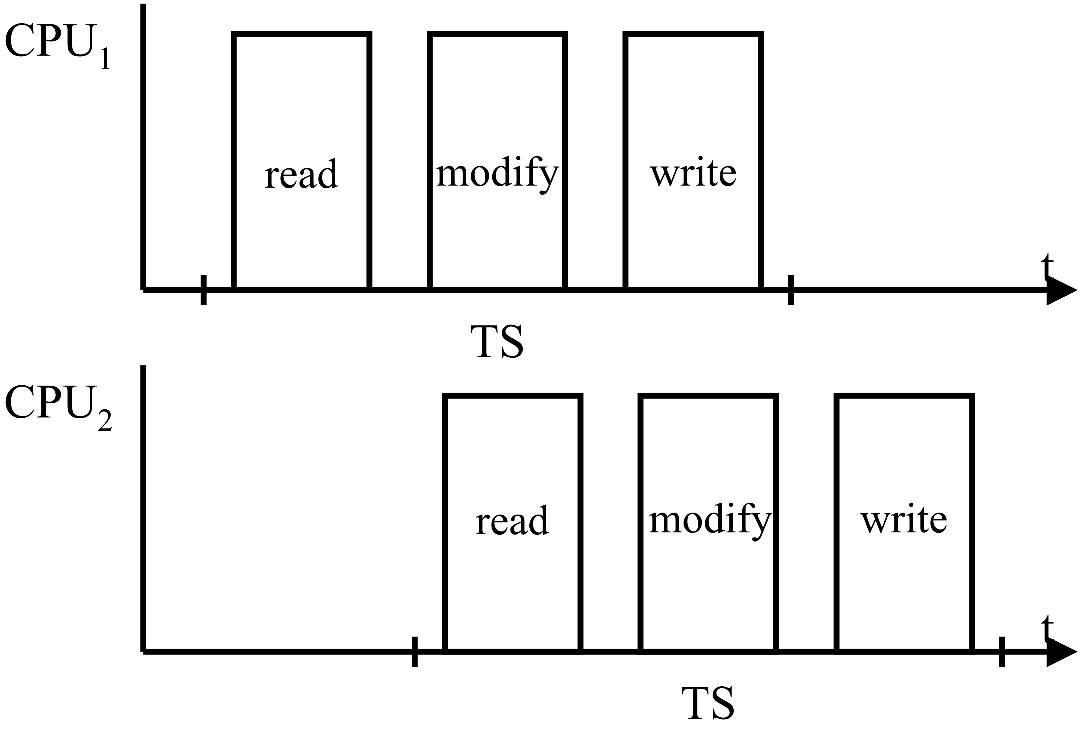
```
P(s): DisableInterrupts;
      P(mutex);
      s := s - 1;
      if s < 0 then
        begin
          Block_Process_Invoking_P_into_L;
          q := Remove_From_RQ;
          V(mutex);
          Transfer_to_q_with_Interrupts_Enabled;
        end
      else begin
        V(mutex);
        EnableInterrupts;
      end;
```

# Avoiding the Busy Wait (cont.)

```
V(s): DisableInterrupts;
      P(mutex);
      s := s + 1;
      if s <= 0 then
        begin
          q := Remove_From_L;
          if there_are_free_CPUs
            then Start_q
            else Add_q_to_RQ;
        end;
      V(mutex);
      EnableInterrupts;
```

A conventional instruction can be used if there is no TS in the CPU instruction set

# TS(x) on Multiprocessor Systems



**Solutions:**

1. Lock memory during TS execution
2. Lock memory with a special prefix instruction

# Monitors
# (Brinch Hansen, 1973. Hoare, 1974)

**The idea:**

Based on the principles of abstract data types.

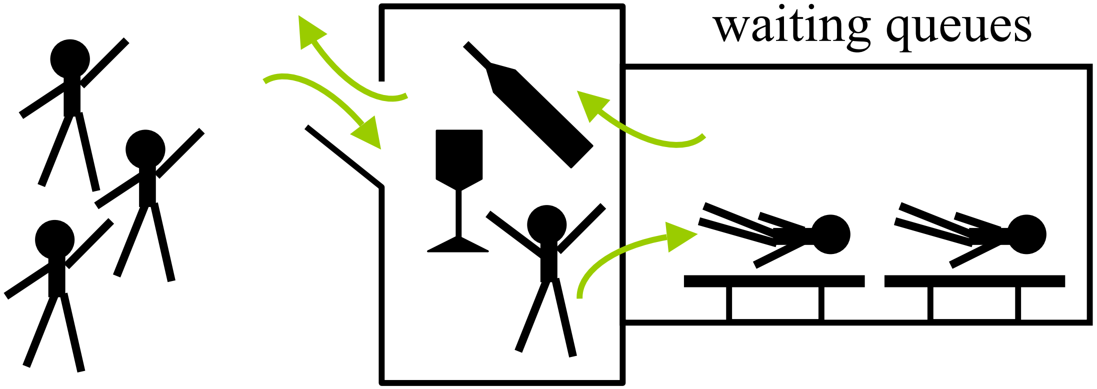
**Monitor:**

1. A set of common resources (variables) and operations (procedures) on them.
2. Procedures are mutually exclusive.
3. Provides a special type of variables called *condition*.
4. Only two operations (*wait* and *signal*) operate on conditions.

# Monitor Operations

- *wait(condition X)*
    - Executing process is suspended (blocked) and placed in a queue associated with *condition X*, monitor becomes "open"
- *signal(condition X)*

  One of the processes (if any) waiting on *condition X* is activated and continues to work in the monitor



# Bounded Buffer

```
type buffer: monitor;
var Buf: array[0..n-1] of char;
    nextin, nextout, count: integer;
    notempty, notfull: condition;

procedure Putdata(data: char);
  begin
    if count = n then wait(notfull);
    Buf[nextin] := data;
    nextin := (nextin + 1) mod n;
    count := count + 1;
    signal(notempty);
  end;
```

# Bounded Buffer (cont.)

```
procedure Getdata(var data: char);
  begin
    if count = 0 then wait(notempty);
    data := Buf[nextout];
    nextout := (nextout + 1) mod n;
    count := count - 1;
    signal(notfull);
  end;

begin
  count := nextput := nextin := 0;
end;
```

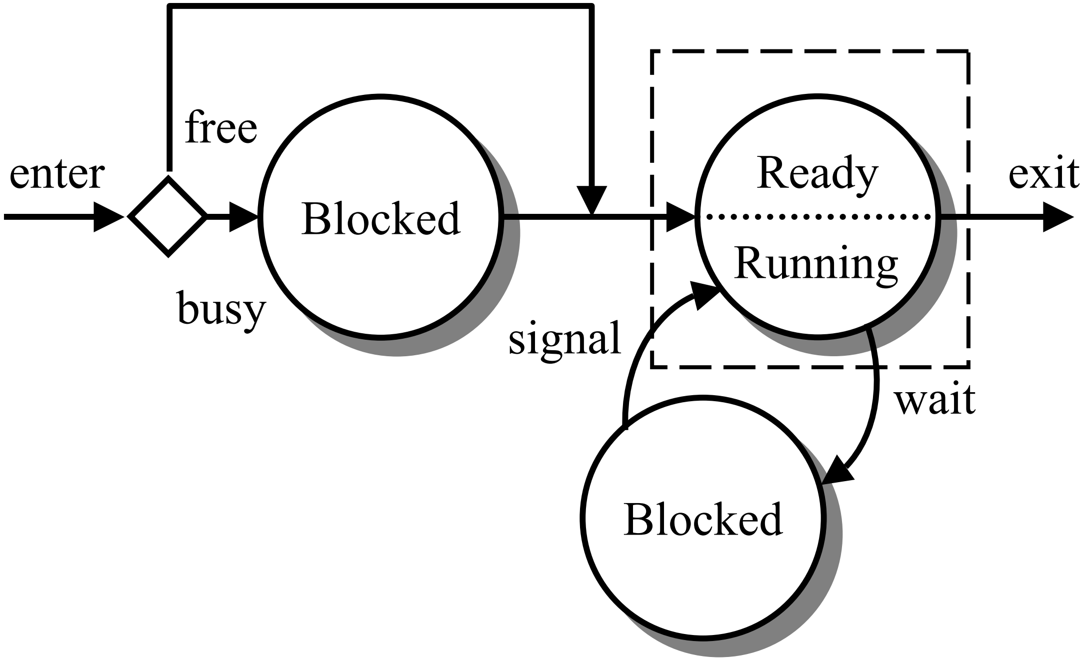
# Bounded Buffer (cont.)

```
var MyBuf: buffer;
```

```
producer$_i$

repeat
  produce_data(data);
  MyBuf.Putdata(data);
until forever;
```

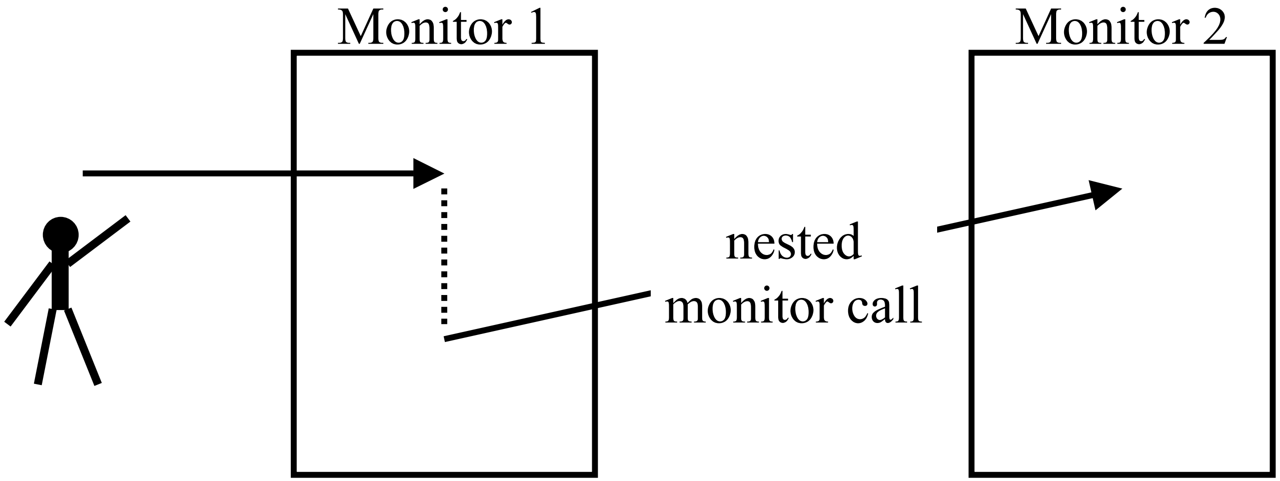
```
consumer$_j$

repeat
  MyBuf.Getdata(data);
  consume_data(data);
until forever;
```

# Diagram of Process States in Monitor



# Problems with Monitors

1. After *signal(condition)* two processes are inside monitor?
2. After a monitor call, if the monitor is busy, the calling process is unconditionally blocked.
3. The problem of nested monitor calls.



# Rendez-vous
# (Hoare and Hansen, 1978)

**The idea:**

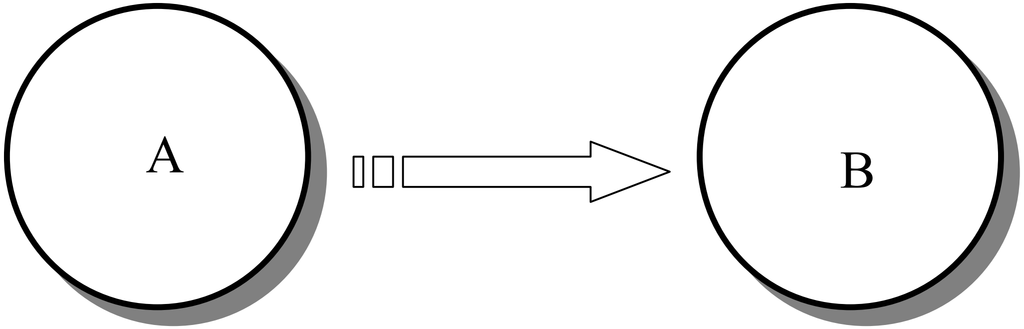
Considers communication and synchronization between processes as inseparable activities.

**The model:**

Process A and process B

A - transmits data

B - receives data



Rendez-vous

# Symmetric Rendez-vous
# (Hoare's model)

Implemented in the Occam programming language

```
process A                          process B
var x: data;                       var y: data;
begin                              begin
   . . .                              . . .
   B!x;          ⇨                    A?y;
   . . .                              . . .
end;                               end;
```

**Disadvantages:**

Every process must know the name of the other process with which it communicates.

For example: we can not build a program library containing processes.

# Asymmetric Rendez-vous

Implemented in the Ada programming language

```
process A           process B
var x: data;        var y: data;
begin               begin
   . . .               . . .
   B.send(x);          accept send ({var}d:data);
   . . .                 y := d;
end;                   end;
                       . . .
                    end;
```

During the execution of *accept* both processes are in rendez-vous.
Operator *accept* is executed as a critical section.

# Asymmetric Rendez-vous (cont.)

**Advantages:**

1. The body of the *accept* operator can be executed from process A as well as from process B.
2. In asymmetric rendez-vous data transmission can be made in both directions.

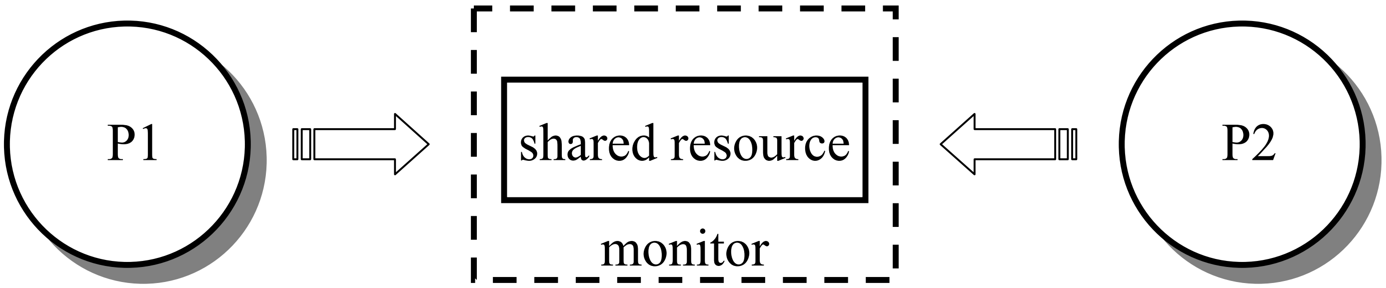
**Disadvantages:**

Model is too simple for realistic tasks.

# Non-deterministic choice of *accept*

```
process Guarded_var;
var shared_var: data;
begin
  repeat;
    select
      accept read(var x: data);
        x := shared_var;
      end;
        or
      accept write(y: data);
        shared_var := y;
      end;
    end select;
  until forever;
end Guarded_var;
```

# Using Rendez-vous for the implementation of mutual exclusion



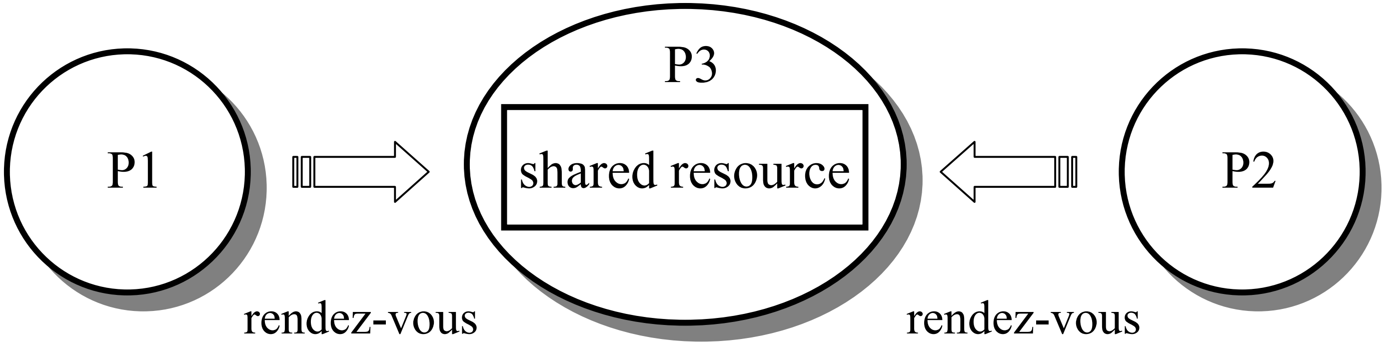
P1

shared resource

monitor

P2

The passive construct **monitor** is replaced with the active construct **process**



P3

P1

shared resource

P2

rendez-vous

rendez-vous

# Rendez-vous disadvantages

In a system using rendez-vous, the number of processes is greater than the number of processes in a system using monitors.

This leads to greater consumption of CPU time for process switching.

# Modula-2

- Concurrent programming in Modula-2 is based on the model of co-routines.
- A co-routine is not declared, instead it is created from a procedure.

**Co-routine creation:**

```
PROCEDURE NEWPROCESS(P:PROC; A:ADDRESS;
         S:CARDINAL;VAR P1: ADDRESS);
```

parameters:

- *P* - procedure from which the new co-routine will be created
- *A, S* - the address and size of the co-routine workspace
- *P1* - holds a new co-routine reference

**Co-routine transfer**:

```
PROCEDURE TRANSFER(VAR P1, P2: ADDRESS);
```

*TRANSFER* suspends the current co-routine (the one that called *TRANSFER*), stores a reference to it in *P1* and resumes the co-routine that *P2* identifies.

# Interrupt Handling

`PROCEDURE IOTRANSFER(VAR P1, P2: ADDRESS; I: CARDINAL);`

parameters:

- *P1, P2* - references to co-routines
- *I* - interrupt vector number

A call to *IOTRANSFER* suspends the current co-routine (the interrupt handler), stores a reference to this co-routine in *P1* and allows the co-routine referenced by *P2* to resume execution. In addition *P1* is “installed” as the handler for the interrupt, specified by *I*.

When this interrupt next occurs, the following actions take place:

1. Current co-routine is suspended.
2. A reference to this co-routine is stored in *P2*.
3. The co-routine referenced by *P1* (the interrupt handler) resumes execution

The “interrupt” is identical to *TRANSFER*.

# Example of Interrupt Handling

```
PROCEDURE InterruptHandler;
(* declarations of local variables *)
BEGIN
  (* initialize local variables *)
  LOOP
    IOTRANSFER(handler, mainProcess, interuptvector);
    (* respond to interrupt *)
  END;
END InterruptHandler;
```

- ✓ Can preemptive process scheduling be implemented using the non-preemptive co-routines of Modula-2 ?