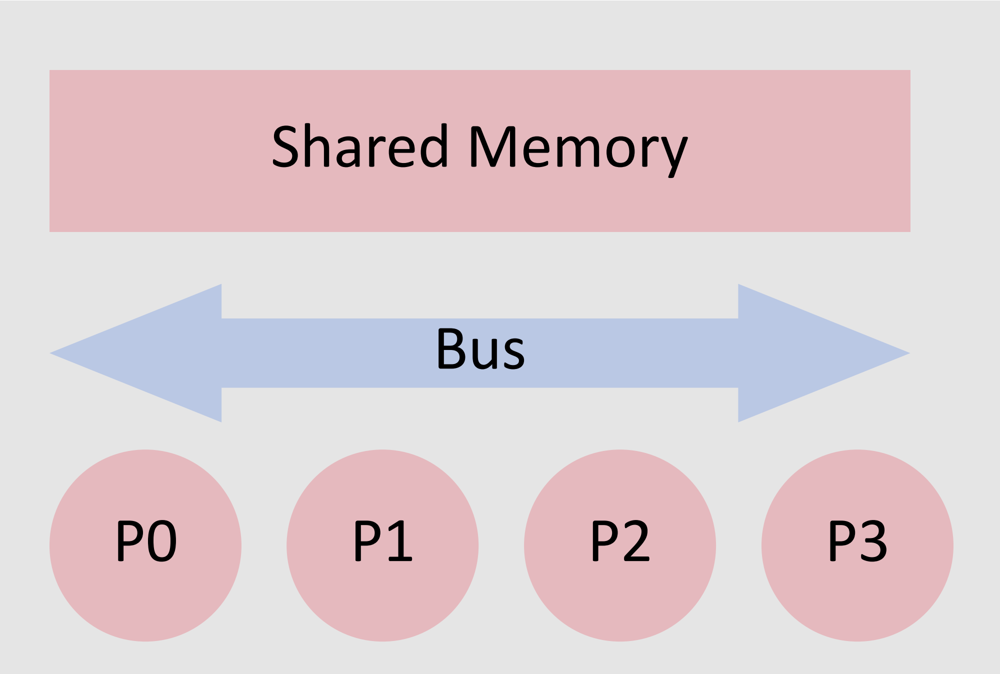
CASE STUDY 2. A2

Architecture



Symmetrical Multiple Processors (SMP)

Useful Resources (x86 compatible HW)

osdev.org: http://wiki.osdev.org

SDM: Intel® 64 and IA-32 Architectures Software Developer’s Manual (4000 p.)

- Vol 1. Architecture
- Vol 2. Instruction Set Reference
- Vol 3. System Programming Guide

MP Spec: Intel Multiprocessor Specification, version 1.4 (100 p.)

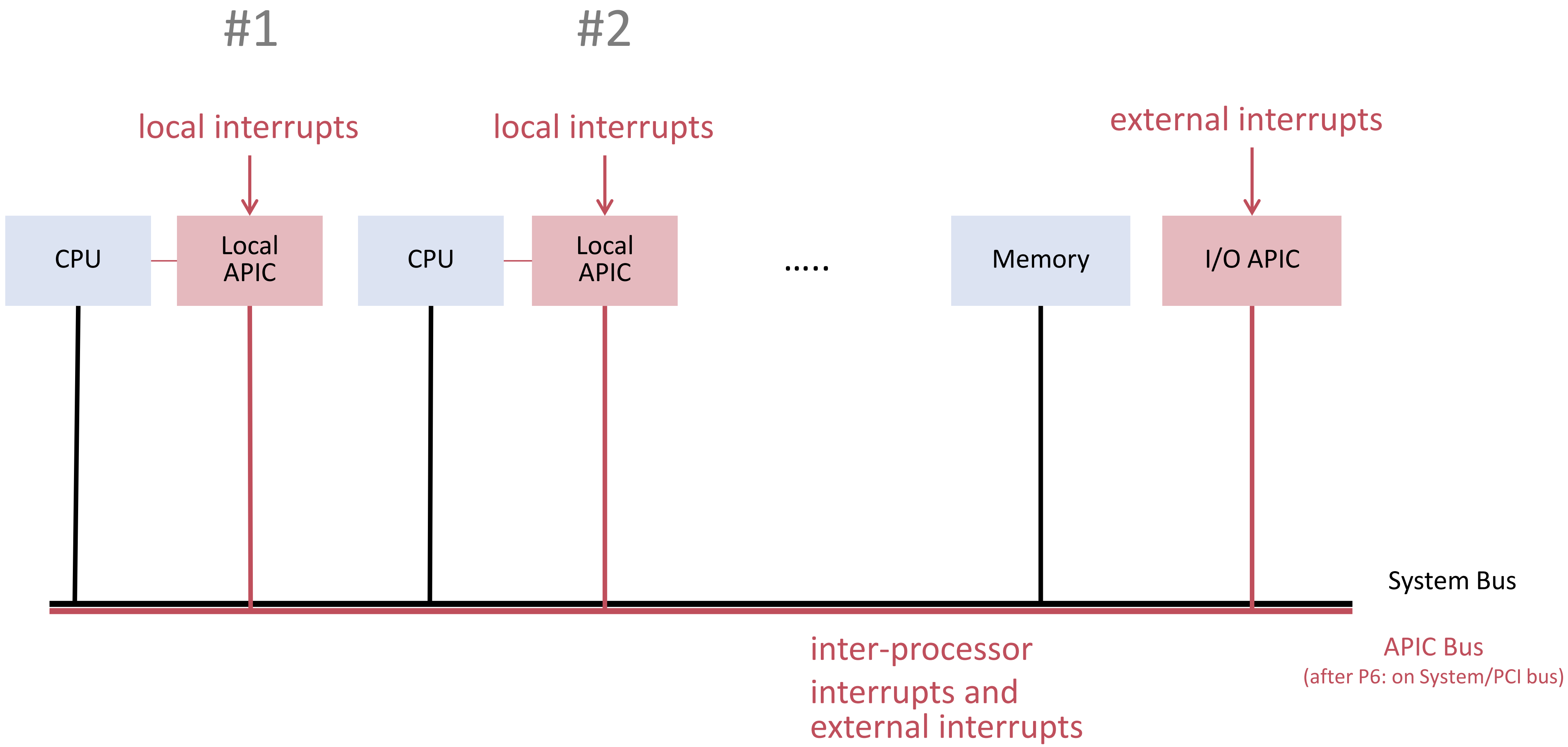
ACPI Spec: Advanced Configuration and Power Interface Specification (1000 p.)

PCI Spec: PCI Local Bus Specification Rev. 2.2 (322 p.)

Interrupt System (x86)

- External interrupts (asynchronous)
  - I/O devices
  - Timer interrupts
  - *Inter-processor interrupts*
- Software interrupts (synchronous)
  - Traps / Syscalls : Special instructions
- Processor exceptions (synchronous)
  - Faults (restartable) – Example: page fault
  - Aborts (fatal) – Example: machine check

APIC Architecture



SDM Vol. 3, Chapter 10

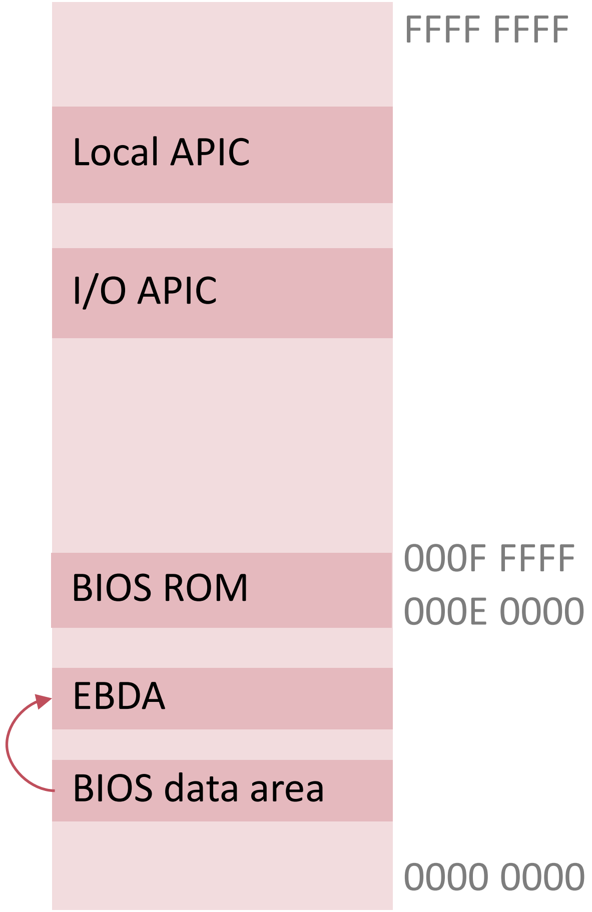
Use of the APIC

- Messages to processors
  - Start Processor
    - Activation and Initialization of individual processors
  - Halt Processor
    - Deactivation of individual processors
  - Halt Process, schedule new process
    - Interrupt in order to transfer control to scheduler
- Local timers
  - Periodical interrupts

MultiProcessor Specification

Standard by Intel (MP Spec 1.4)

- Hardware Specification
  - Memory Map
  - APIC
  - Interrupt Modes
- MP Configuration Table
  - Processor, Bus, I/O APIC
  - Table address searched via "floating pointer structure"



MP Spec

Other configuration methods

Local APIC address ← RDMSR instruction

Check presence of APIC and MSR via CPUID instruction

- Local APIC register region must be mapped strong uncacheable

IO APIC address ←ACPI table

Advanced Configuration and Power Interface Specification

- configuration table
- AML code

SDM
ACPI Spec

Exception Numbers

Vector #	Description	Source
0	Div error	`div` / `idiv` instruction
1	Debug	Code / data reference
2	NMI	Non maskable external IRQ
3	Breakpoint	`int 3` instruction
4 – 19	Other processor exceptions	E.g. page fault etc.
20-31	reserved	
32-255	Maskable Interrupts	External Interrupts from INTR pin INT n instruction

Configuring APIC

- Local Vector Table
  - Vector Number, Trigger Mode, Status, Interrupt Vector Mask
  - Timer Mode (one shot / periodic)

for local interrupt sources

- Command Register: Inter Processor Interrupt with
  - vector number,
  - delivery mode: fixed, nmi, init, startup (..)
  - logical / physical destination
    (including self and broadcasts with / without self)

PCI Local Bus

Peripheral Component Interconnect Specification

- Standardized Configuration Address Space for all PCI Devices
- Interrupt Routing Configuration

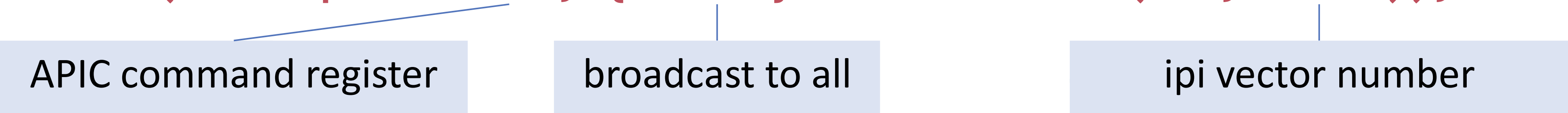
Access Mechanisms

- PCI BIOS – offers functionality such as "find device by classcode"
  Presence determined by floating data structure in BIOS ROM
- Addressable via in / out instructions operating on separate I/O memory address space

register (offset)	bits 31-24	bits 23-16	bits 15-8	bits 7-0
00	Device ID		Vendor ID	
04	Status		Command	
08	Class code	Subclass	Prog IF	Revision ID
0C	BIST	Header type	Latency Timer	Cache Line Size
10	Base address #0 (BAR0)			
14	Base address #1 (BAR1)			
...				
3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line
...				

Broadcast an operation

```
(** Broadcast an operation to all processors. *)
PROCEDURE Broadcast* (h: BroadcastHandler; msg: Message; flags: SET);
BEGIN
 Acquire(Processors);
 ipcBusy := allProcessors;
 ipcHandler := h; ipcMessage := msg; ipcFlags := flags;

 SYSTEM.PUT(localApic + 300H, {18..19} + SYSTEM.VAL (SET, MPIPC));
```



APIC command register

broadcast to all

ipi vector number

```
 WHILE ipcBusy # {} DO SpinHint END;

 Release(Processors)
END Broadcast;
```

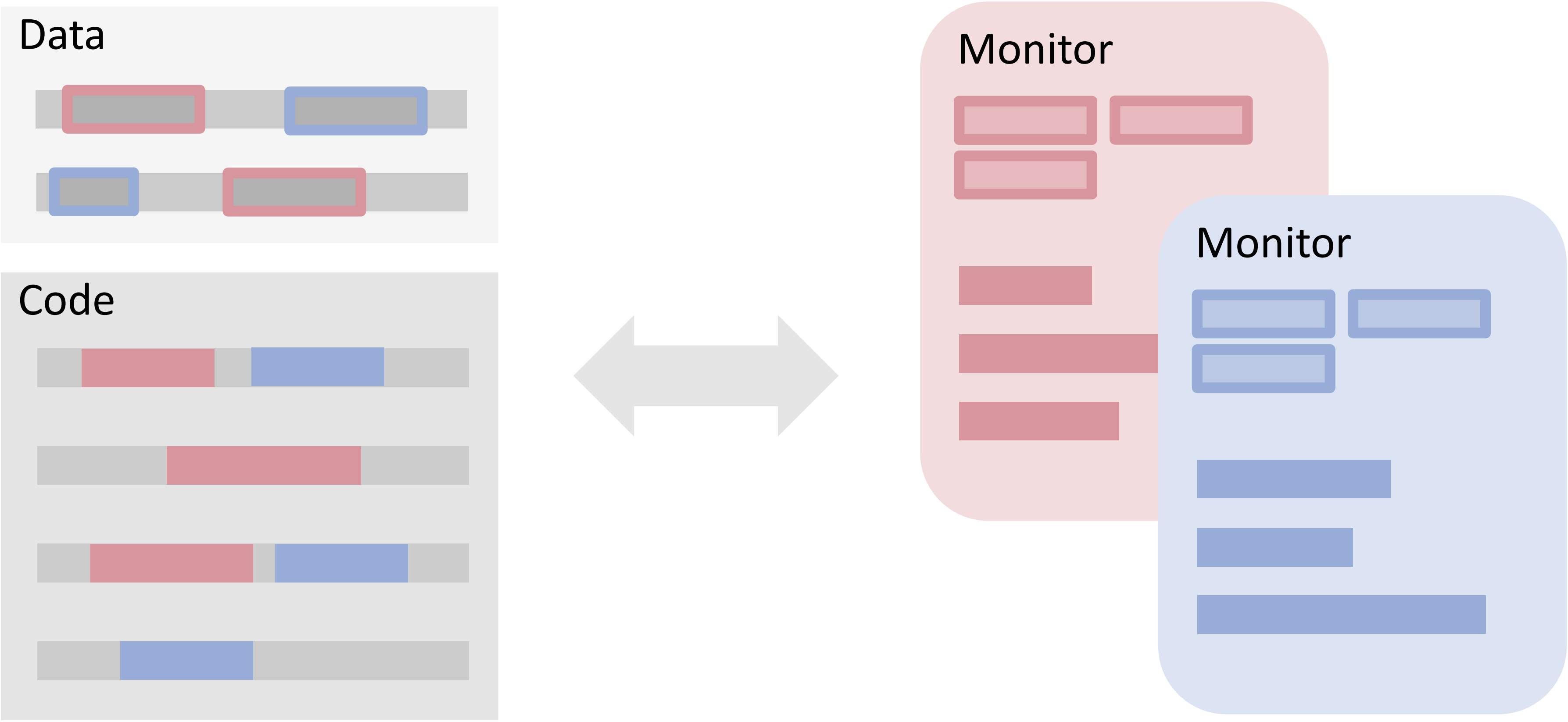
Handling the IPI

```
(* Handle interprocessor interrupt.
  Interrupts are off and processor is at kernel level. *)
PROCEDURE HandleIPC(VAR state: State);
VAR id: LONGINT;
BEGIN
  id := ID();

  ipcHandler(id, state, ipcMessage);(* interrupts off and at kernel level *)

  AtomicExcl(ipcBusy, id) (* ack *)

  IF state.INT = MPIPC THEN ApicPut(0B0H, {})  END
END HandleIPC;
```

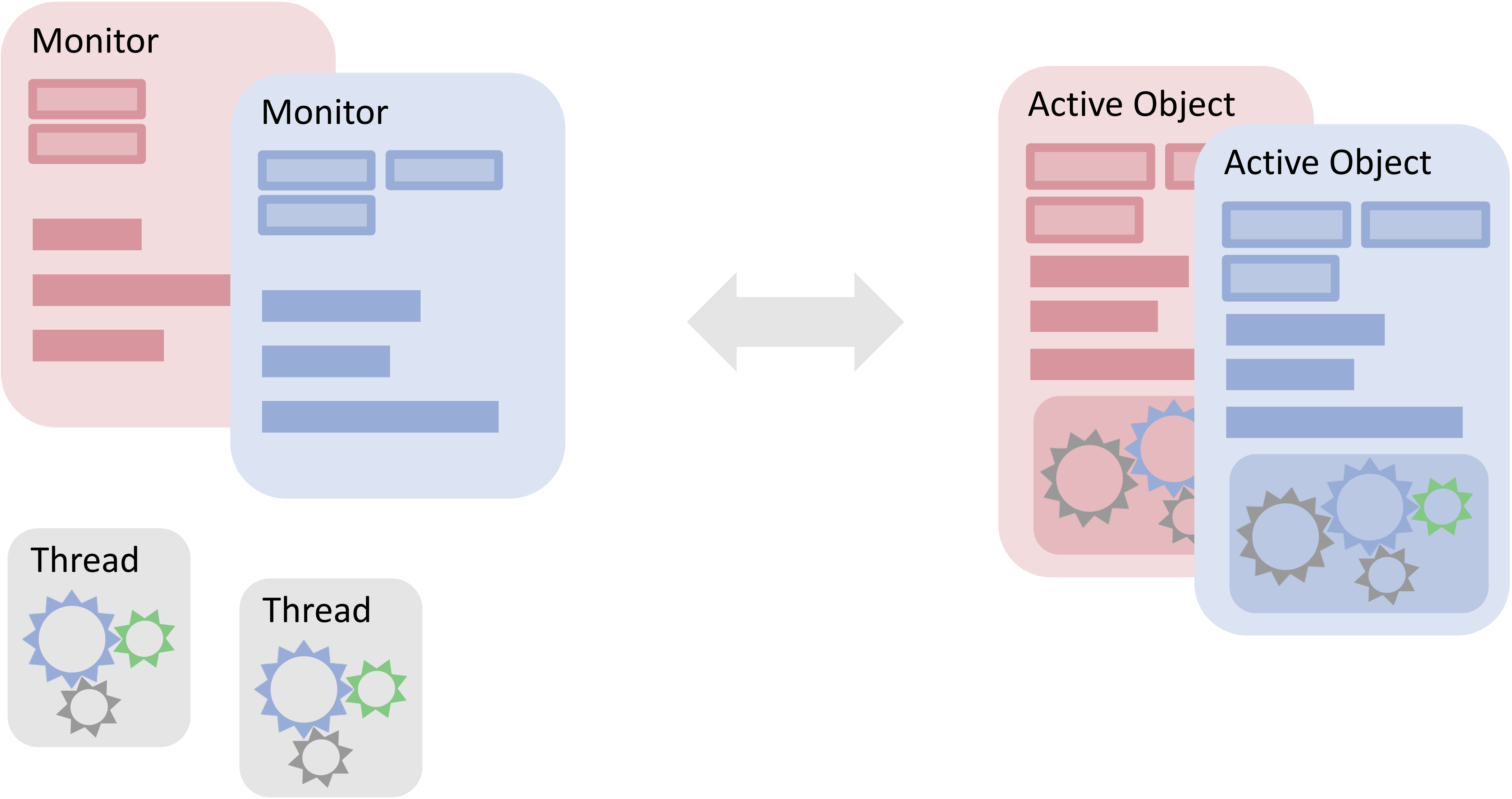
EOI register

2.1. ACTIVE OBERON LANGUAGE

Locks vs. Monitors

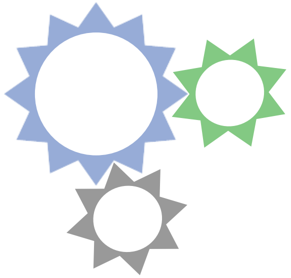


Threads vs. Active Objects



Object Model

```
TYPE
   MyObject = OBJECT
   VAR i: INTEGER; x: X;

      PROCEDURE & Init (a, b: X);
      BEGIN... (* initialization *) END Init;

      PROCEDURE f (a, b: X): X;
      BEGIN{EXCLUSIVE}
         ...
         AWAIT i >= 0;
         ...
      END f;

   BEGIN{ACTIVE}
      ...
      BEGIN{EXCLUSIVE}
         i := 10; ....
      END ...
   END MyObject;
```



Protection
Methods tagged **exclusive** run under mutual exclusion

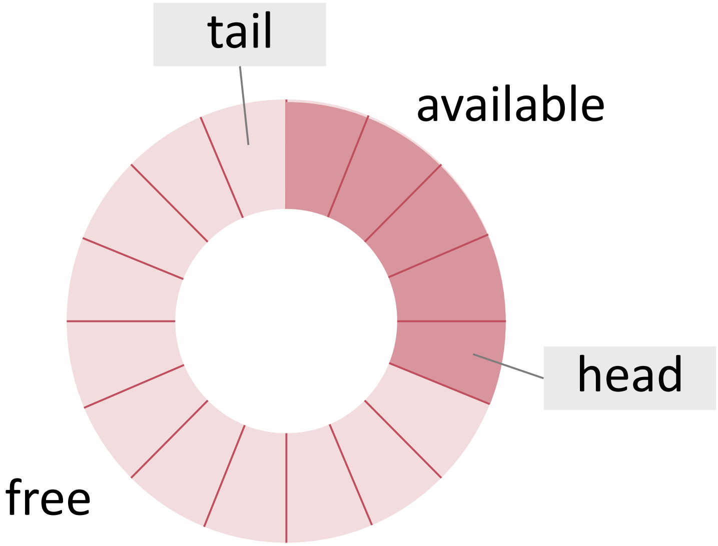
Synchronisation
Wait until condition of **await** becomes true

Parallelism
Body marked **active** executed as thread for each instance

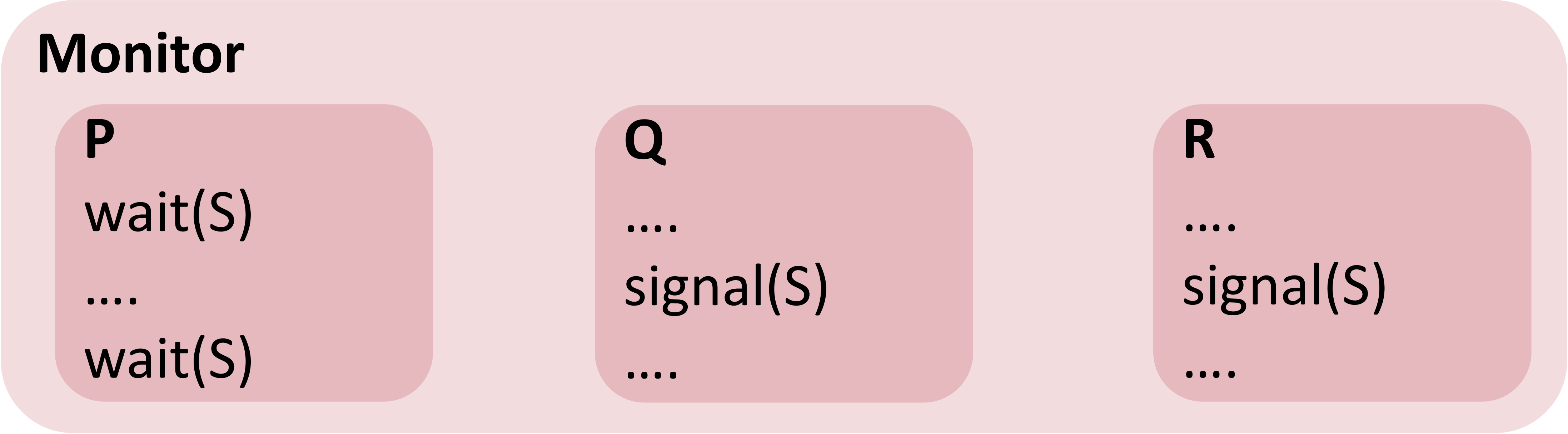
The **await** Construct

```
VAR
head, tail, available, free: INTEGER;
buf: ARRAY N of object;

PROCEDURE Produce (x: object);
BEGIN{EXCLUSIVE}
  AWAIT(free # 0);
  DEC(free); buf[tail] := x;
  tail := (tail + 1) mod N;
  INC(available);
END Produce;
```

```
PROCEDURE Consume (): object;
  VAR x: object;
BEGIN{EXCLUSIVE}
  AWAIT(available # 0);
  DEC(available); x := buf[head];
  head := (head + 1) MOD N;
  INC(free); RETURN x
END Consume;
```



Signal-Wait Scenario



Monitor

P

```
wait(S)
….
wait(S)
```

Q

```
….
signal(S)
….
```

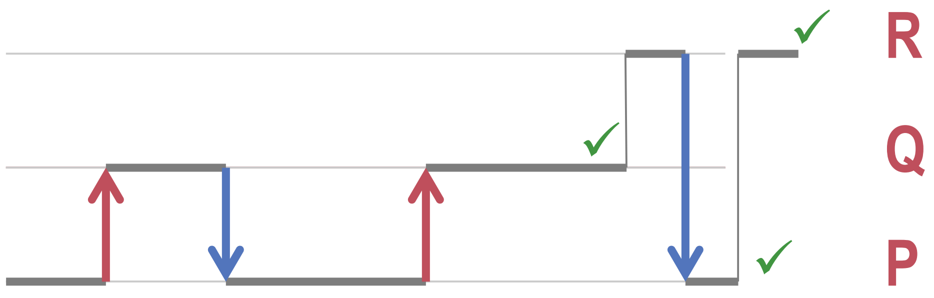
R

```
….
signal(S)
….
```

Signal-Wait Implementations

“Signal-And-Pass”

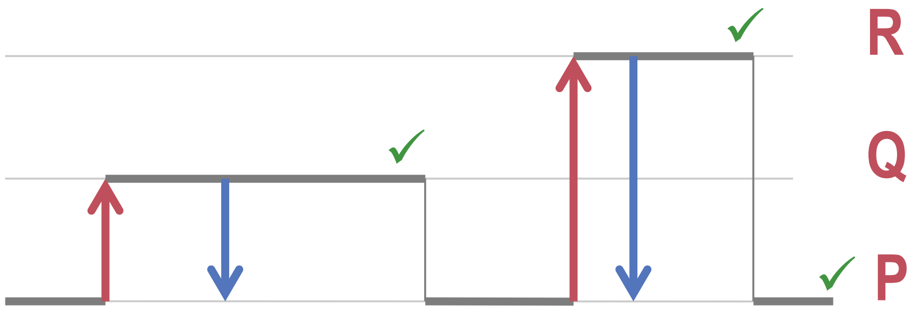
(aka Signal and Wait)



wait

signal

“Signal-And-Continue”



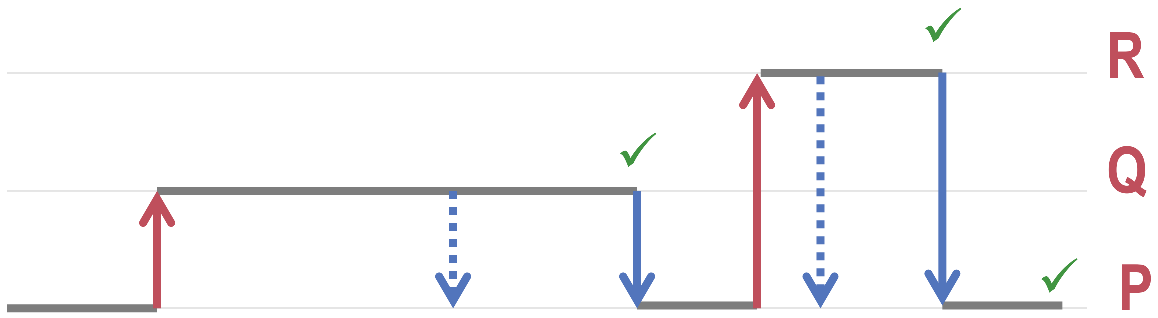
end

less context switches

Signal-Wait Implementations

“Signal-And-Exit”

(await queues have priority)



Await c

c true

signal

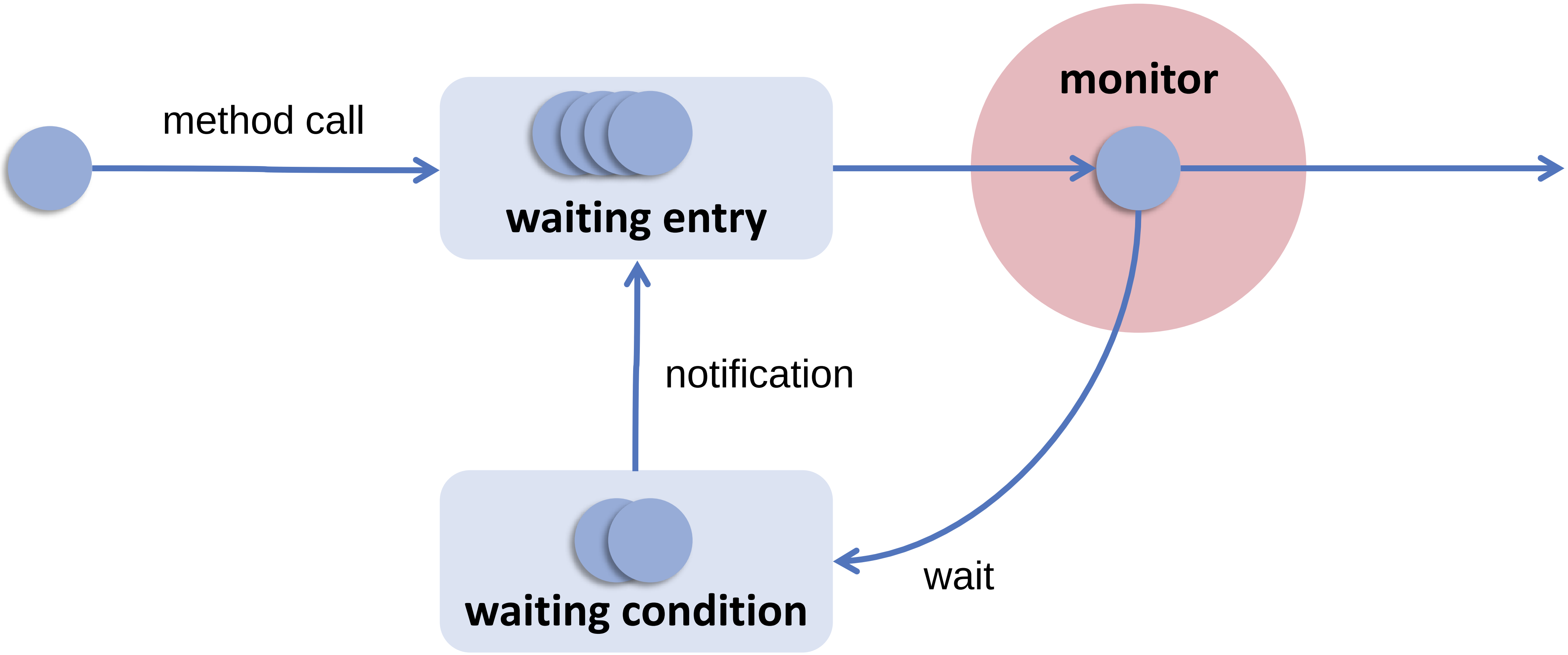
current implementation in Active Oberon

Why this is important? Let's try this:

```
class Semaphore{
   int number = 1; // number of threads allowed in critical section

   synchronized void enter() {
      if (number <= 0)
         try { wait();  } catch (InterruptedException e) { };
      number--;
   }

   synchronized void exit() {
      number++;
      if (number > 0)
         notify();
   }
}
```

Looks good, doesn't it?
But there is a problem.
Do you know?

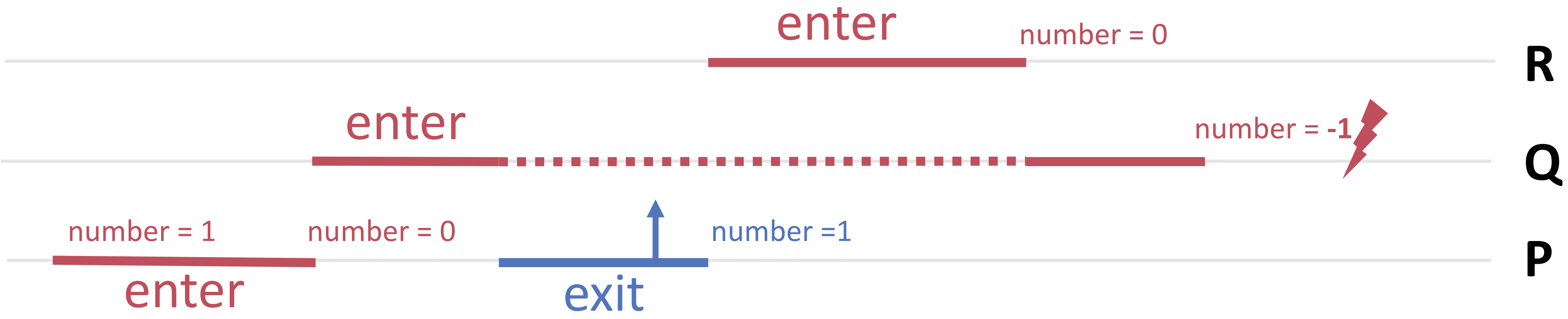
Java Monitor Queues



Java Monitors = signal + continue

```
synchronized void enter() {
   if (number <= 0)
      try { wait();   }
      catch (InterruptedException e) {
};
   number--;
}
```

```
synchronized void exit() {
   number++;
   if (number > 0)
      notify();
}
```



The cure.

```
synchronized void enter() {
   while (number <= 0)
      try { wait();   }
      catch (InterruptedException e) { };
   number--;
}
```

```
synchronized void exit()
{
   number++;
   if (number > 0)
      notify();
}
```

If, additionally, different threads evaluate different conditions, the notification has to be a `notifyAll`. In this example this is not required.

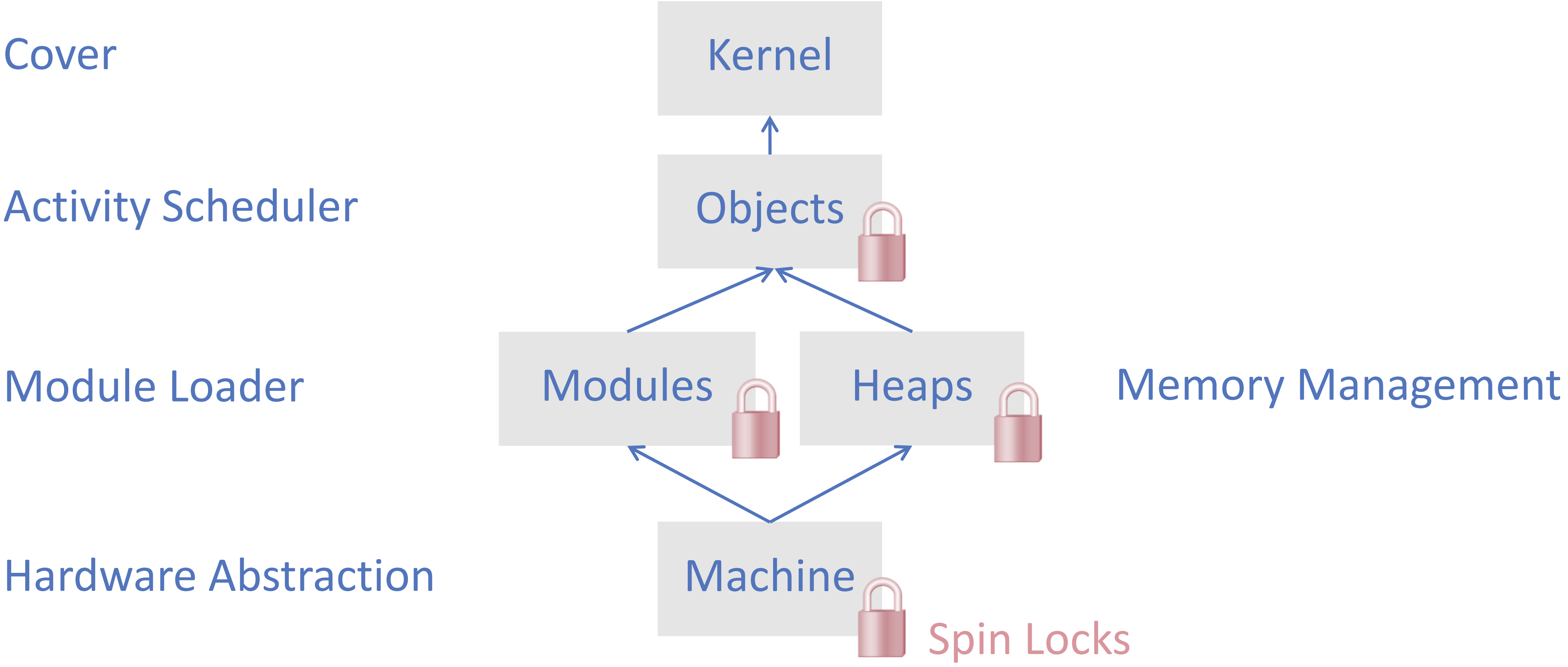
(In Active Oberon)

```
Semaphore = object
    number := 1: longint;

    procedure enter;
    begin{exclusive}
        await number > 0;
        dec(number)
    end enter;

    procedure exit;
    begin{exclusive}
        inc(number)
    end exit;

end Semaphore;
```

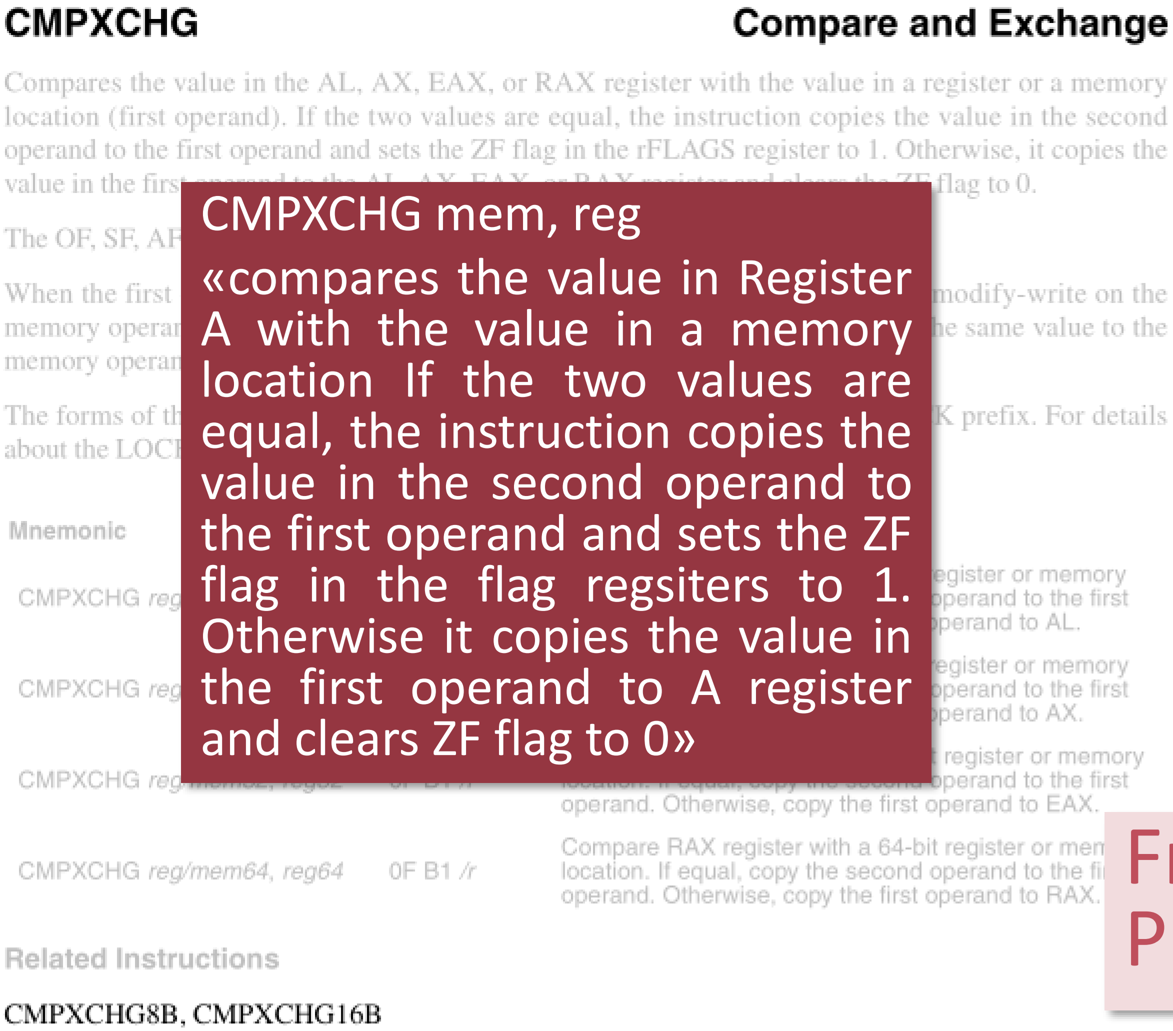
```
class Semaphore{
    int number = 1;

    synchronized void enter() {
        while (number <= 0)
            try { wait();}
            catch (InterruptedException e) { };
        number--;
    }

    synchronized void exit() {
        number++;
        if (number > 0)
            notify();
    }
}
```

2.2. ACTIVE OBJECT SYSTEM (A2)

Modular Kernel Structure



Hardware support for atomic operations: Example



CMPXCHG **Compare and Exchange**

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF

When the first … modify-write on the memory operand … he same value to the memory operand

The forms of th … K prefix. For details about the LOCK

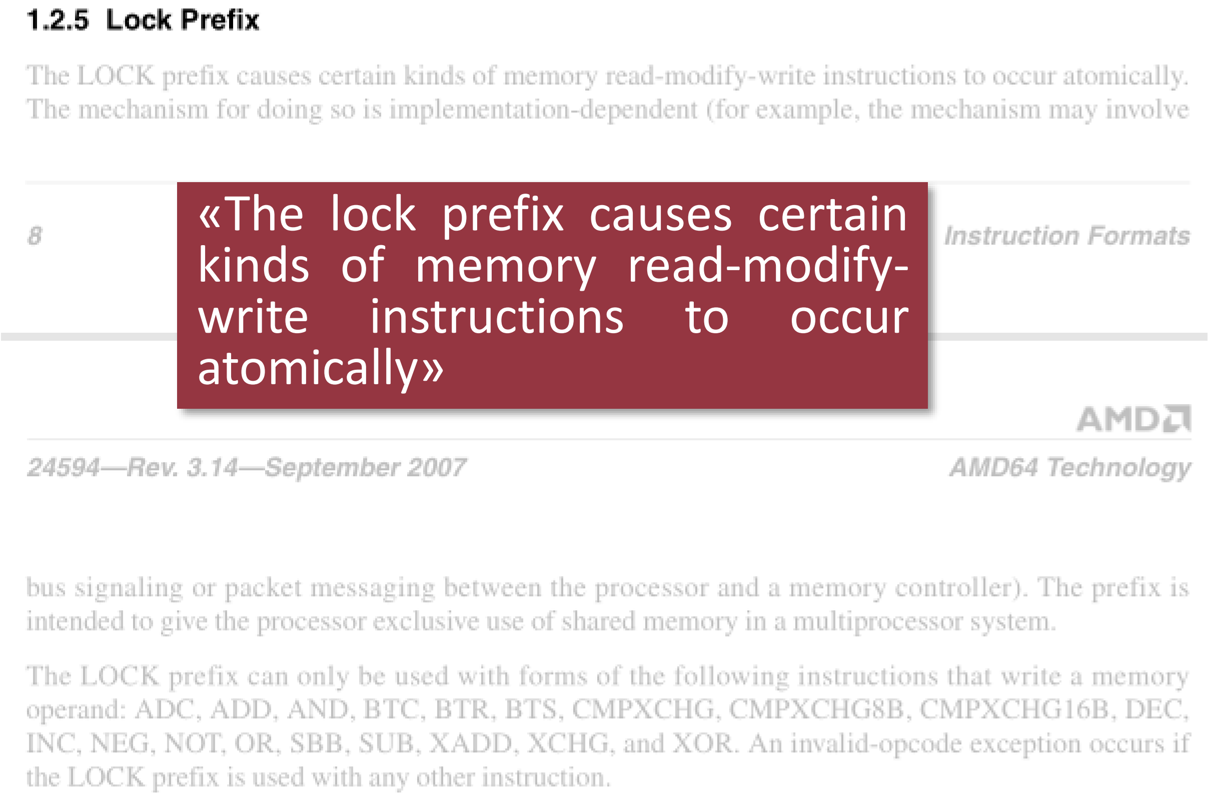
Mnemonic		
CMPXCHG reg		egister or memory operand to the first operand to AL.
CMPXCHG reg		egister or memory operand to the first operand to AX.
CMPXCHG reg/mem32, reg32	0F B1 /r	register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to EAX.
CMPXCHG *reg/mem64*, *reg64*	0F B1 */r*	Compare RAX register with a 64-bit register or mem location. If equal, copy the second operand to the fi operand. Otherwise, copy the first operand to RAX.

Related Instructions

CMPXCHG8B, CMPXCHG16B

CMPXCHG mem, reg

«compares the value in Register A with the value in a memory location If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag regsiters to 1. Otherwise it copies the value in the first operand to A register and clears ZF flag to 0»



1.2.5 Lock Prefix

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve

8 *Instruction Formats*

AMD

24594—Rev. 3.14—September 2007 *AMD64 Technology*

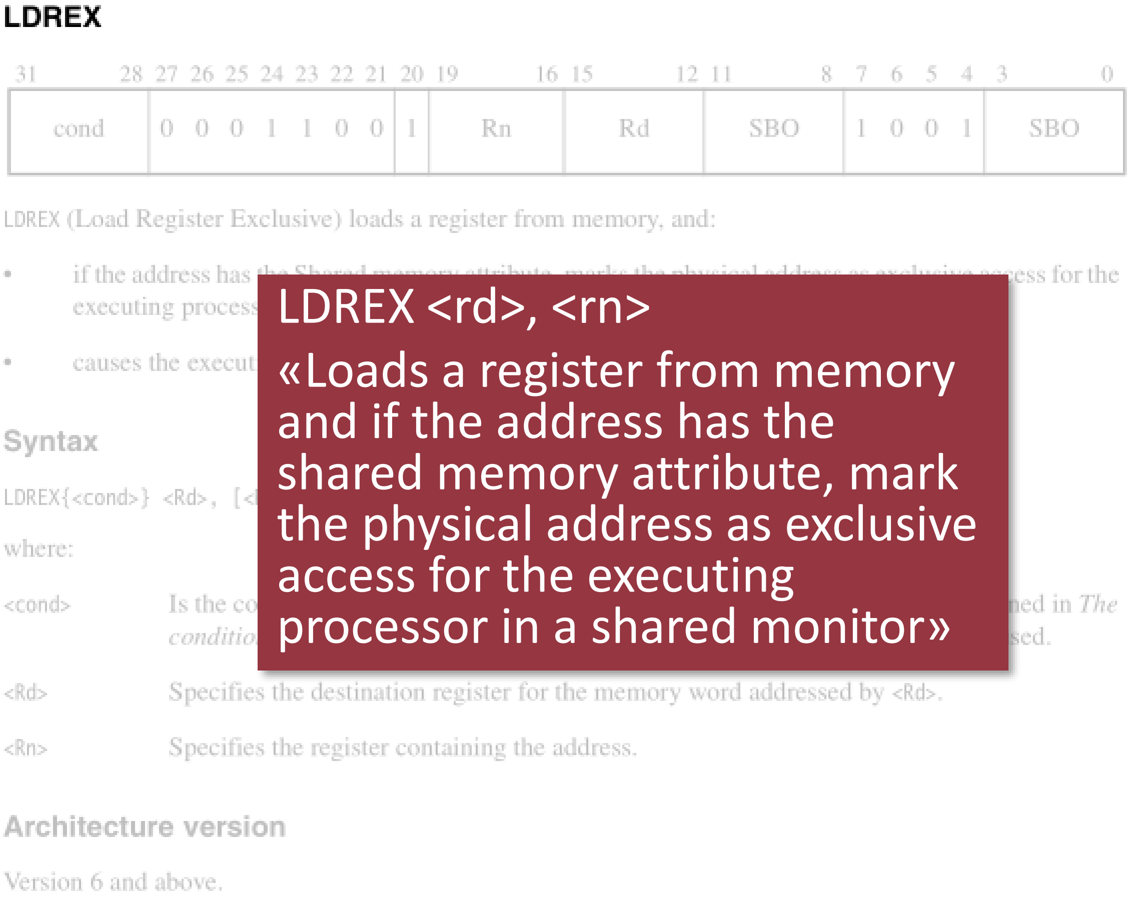
bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

«The lock prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

From AMD64 Architecture Programmer's Manual

Hardware support for atomic operations: Example



LDREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	1	Rn	Rd	SBO	1 0 0 1	SBO

LDREX (Load Register Exclusive) loads a register from memory, and:

- if the address has the Shared memory attribute, marks the physical address as exclusive access for the executing process
- causes the execut

Syntax

LDREX{<cond>} <Rd>, [<

where:

<cond> Is the co ned in *The conditio* sed.

<Rd> Specifies the destination register for the memory word addressed by <Rd>.

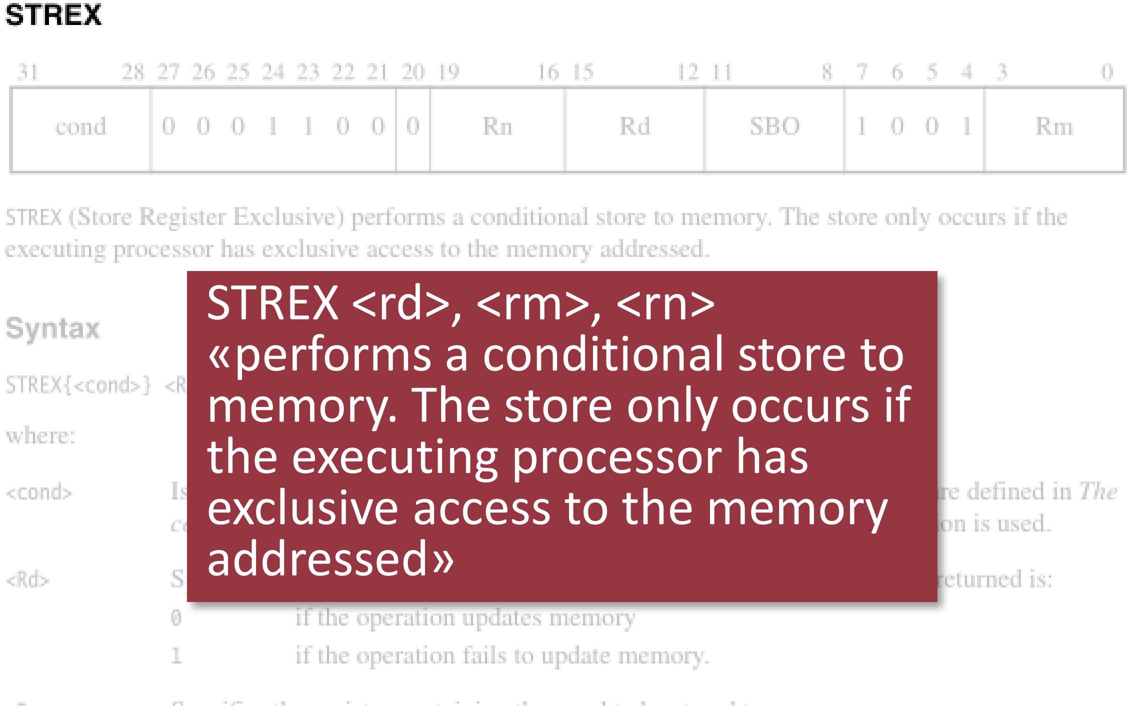
<Rn> Specifies the register containing the address.

Architecture version

Version 6 and above.

LDREX <rd>, <rn>

«Loads a register from memory and if the address has the shared memory attribute, mark the physical address as exclusive access for the executing processor in a shared monitor»



STREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	0	Rn	Rd	SBO	1 0 0 1	Rm

STREX (Store Register Exclusive) performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

STREX{<cond>} <R

where:

<cond> Is re defined in *The* on is used.

<Rd> S returned is:

0 if the operation updates memory

1 if the operation fails to update memory.

STREX <rd>, <rm>, <rn>
«performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed»

From ARM Architecture Reference Manual

Hardware support for atomic operations

Typical instructions:

- Test-And-Set (TAS),
    - Example TSL register,flag (Motorola 68000)
- Compare-And-Swap (CAS).
    - Example: LOCK CMPXCHG (Intel x86)
    - Example: CASA (Sparc)
- Load Linked / Store Conditional.
    - Example LDREX/STREX (ARM),
    - Example LL / SC (MIPS)

typically several orders of magnitute slower than simple read & write operations !

TAS Semantics

TAS(var s: word): boolean;

atomic

```
if (s == 0) then
        s := 1;
        return true;
else
        return false;
end;
```

Implementation of a spinlock using TAS

```
Init(var lock: word);
    lock := 0;
```

```
Acquire (var lock: word)
    repeat until TAS(lock);
```

```
Release (var lock: word)
    lock = 0;
```

CAS Semantics

```
CAS (var a:word, old, new: word): word;
    oldval := a;
    if (old = oldval) then
        a := new;
    end;
    return oldval;
```

atomic

Implementation of a spinlock using CAS

Init(lock)

```
    lock = 0;
```

Acquire (var lock: word)

```
    repeat
        res := CAS(lock, 0, 1);
    until res = 0;
```

Release (var lock: word)

```
    CAS(lock, 1, 0);
```

API *Machine*

implemented by
I386.Machine.Mod, AMD64.Machine.Mod, Win32.Machine.Mod, Unix.Machine.Mod

```
MODULE Machine;
  TYPE
   State* = RECORD (*processor state*) END;
   Handler* = PROCEDURE {DELEGATE}(VAR state: State);

   PROCEDURE ID* (): LONGINT;

   PROCEDURE AcquireObject(VAR locked: BOOLEAN);
   PROCEDURE ReleaseObject(VAR locked: BOOLEAN);

   PROCEDURE Acquire*(level: LONGINT);
   PROCEDURE Release*(level: LONGINT);

   // paging support
   // irq support
END Machine.
```

Low level locks

Processor management

Virtual Memory Management

IRQs

API *Heaps*

conceptually portable

```
MODULE Heaps;

TYPE
(* base object of heap blocks *)
HeapBlock* = POINTER TO HeapBlockDesc;
HeapBlockDesc* = RECORD … END;
RecordBlock* = POINTER TO RecordBlockDesc;
RecordBlockDesc = RECORD  (HeapBlockDesc) END;

Finalizer* = PROCEDURE {DELEGATE} (obj: ANY);
FinalizerNode* = POINTER TO RECORD
   objWeak* {UNTRACED}: ANY;   (* weak reference to checked object *)
   objStrong*: ANY; (* strong reference to object to be finalized *)
   finalizer* {UNTRACED} : Finalizer;
END;

PROCEDURE AddFinalizer*(obj: ANY; n: FinalizerNode);
PROCEDURE GetHeapInfo*(VAR total, free, largest: SYSTEM.SIZE)

Procedures NewSys*, NewRec*, NewProtRec*, NewArr*
```

Heap Management
Allocation
Garbage Collector
Finalizers

API *Modules*

portable

```
MODULE Modules;
  TYPE
   Module* = OBJECT (*module data*) END Module;

   PROCEDURE ThisModule*(CONST name: ARRAY OF CHAR;
     VAR res: LONGINT;
     VAR msg: ARRAY OF CHAR): Module;

   PROCEDURE FreeModule*(CONST name: ARRAY OF CHAR;
     VAR res: LONGINT; VAR msg: ARRAY OF CHAR);

   PROCEDURE InstallTermHandler*
     (h: TerminationHandler); (*called when freed*)

   PROCEDURE Shutdown*(Mcode: LONGINT); (*free all*)

END Modules.
```

Module Loader
Loading
Unloading
Termination Handlers

API *Objects*

conceptually portable

```
MODULE Objects;
  TYPE
   EventHandler* = PROCEDURE {DELEGATE};

   PROCEDURE Yield*; (* to other processes *)

   PROCEDURE ActiveObject* (): ANY; (* current process *)

   PROCEDURE SetPriority* (p: LONGINT); (*for current*)

   PROCEDURE InstallHandler* (h: EventHandler; int: LONGINT);

   PROCEDURE RemoveHandler*(h: EventHandler; int: LONGINT);

   Procedures CreateProcess, Lock, Unlock, Await

END Objects.
```

Scheduler
Timer Interrupt
Process Synchronisation
2nd Level Interrupt Handlers

API *Kernel*

conceptually portable

```
MODULE Kernel;

  PROCEDURE GC*; (* activate garbage collector*)

  TYPE
   Timer* = OBJECT (*delay timer*);
     PROCEDURE Sleep*(ms: LONGINT);
     PROCEDURE Wakeup*;
  END Timer;

   FinalizedCollection*=OBJECT

       PROCEDURE Add*(obj: ANY; fin: Finalizer);
       PROCEDURE Remove*(obj: ANY);
       PROCEDURE Enumerate*(enum: Enumerator);

END Kernel.
```

Kernel Cover

Boot Procedure

- Start BIOS Firmware
- Load A2 Bootfile
- Initialize modules
  - Module *Machine*
  - Module *Heaps*
  - …
  - Module Objects
    - Setup scheduler and self process
  - Module *Kernel*
    - Start all processors
  - …
  - Module Bootconsole
    - read configuration and execute boot commands

BP (boot processor)

Processor Startup

- Start processor P (Bootprocessor)
  1. Setup boot program
  2. Enter processor IDs into table
  3. Send *startup* message to P via APIC
  4. Wait with timeout on *started* flag by P

Machine.InitProcessors,
Machine.InitBootPage

Machine.ParseMPConfig

Machine.StartProcessor

- Boot program (For each processor)
  1. Set 32-bit runtime environment
  2. Initialize control registers, memory management, interrupt handling, APIC
  3. Set *started* flag
  4. Setup Scheduler

for all processors

Machine.EnterMP

Machine.StartMP

Objects.Start

5. Bootprocessor proceeds with boot console