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

**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.)
APIC Architecture

System Bus

APIC Bus
(after P6: on System/PCI bus)

I/O Apic -> Local APIC
IPIs

inter-processor interrupts and external interrupts

CPU Local APIC CPU Local APIC Memory I/O APIC

local interrupts local interrupts external interrupts

apic timer, thermal sensor, performance monitoring,

external devices

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"
Other configuration methods

Local APIC address \( \leftrightarrow \) RDMSR instruction

Check presence of APIC and MSR via CPUID instruction

- Local APIC register region must be mapped strong uncacheable

IO APIC address \( \leftrightarrow \) ACPI table

Advanced Configuration and Power Interface Specification

- configuration table
- AML code
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
2.1. ACTIVE OBERON LANGUAGE
Threads vs. Active Objects

Monitor

Thread

Monitor

Thread

Active Object

Active Object
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;
BEGIN{EXCLUSIVE}
  AWAIT(available # 0);
  DEC(available); x := buf[head];
  head := (head + 1) MOD N;
  INC(free); RETURN x
END Consume;
### Signal-Wait Scenario

<table>
<thead>
<tr>
<th>Monitor</th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wait(S)</td>
<td>.... signal(S)</td>
<td>.... signal(S)</td>
</tr>
<tr>
<td></td>
<td>.... wait(S)</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>
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)

c 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

- Method call
- Waiting entry
- Notification
- Waiting condition
- Wait
- Monitor
Java Monitors = signal + continue

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

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

```java
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: integer;

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

Cover

Activity Scheduler

Module Loader

Hardware Abstraction

Kernel

Objects

Modules

Heaps

Machine

Memory Management

Spin Locks
Hardware support for atomic operations: Example

**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 register to 1. Otherwise, it copies the value in the first operand to A register and clears ZF flag to 0.»

«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 <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 <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 magnitude slower than simple read & write operations!**
TAS Semantics

TAS(var s: word): boolean;

if (s == 0) then
    s := 1;
    return true;
else
    return false;
end;
Implementation of a spinlock using TAS

\textbf{Init (var lock: word)};

\hspace{1cm} lock := 0;

\textbf{Acquire (var lock: word)}

\hspace{1cm} repeat until TAS(lock);

\textbf{Release (var lock: word)}

\hspace{1cm} lock = 0;
CAS Semantics


CAS (var a:word, old, new: word): word;

oldval := a;
if (old = oldval) then
  a := new;
end;
return oldval;
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

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

PROCEDURE ID* (): SIZE;

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

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

// paging support
// irq support
END Machine.
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*
API Modules

portable

MODULE Modules;

TYPE
Module* = OBJECT (*module data*) END Module;

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

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

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

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

END Modules.
API Objects
conceptually portable

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

  PROCEDURE Yield*; (* to other processes *)

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

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

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

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

  Procedures CreateProcess, Lock, Unlock, Await

END Objects.

Scheduler
Timer Interrupt
Process Synchronisation
2nd Level Interrupt Handlers
MODULE Kernel;

PROCEDURE GC*; (* activate garbage collector*)

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

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

END Kernel.
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
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

- 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
  5. Bootprocessor proceeds with boot console

2.3. ACTIVITY MANAGEMENT
Life Cycle of Activities

- Running
- Awaiting Condition
- Awaiting Lock
- Ready
- Terminated
- NIL
Runtime Data Structures

- **running array**: Processes 1, 2, 3, 4 mapped to processor IDs 1, 2, 3, 4.
- **object header**: Cond and lock fields for per (monitor) object.
- **ready queues array**: Idle, low, medium, high, GC, RT priority levels.
- **interrupt array**: IRQs 0, 1, 2, 3, 4, 5.
Process Descriptors

TYPE
  Process = OBJECT
  ...
    stack: Stack;
    state: ProcessState;
    preempted: BOOLEAN;
    condition: PROCEDURE (slink: ADDRESS);
    conditionFP: ADDRESS;
    priority: INTEGER;
    obj: OBJECT;
    next: Process
END Process;

ProcessQueue = RECORD
  head, tail: Process
END;

VAR
  ready: ARRAY NumPriorities OF ProcessQueue;
  running: ARRAY NumProcessors OF Process;
Process Dispatching

PROCEDURE Enter (p: Process);
BEGIN
  Put(ready[p.priority], p);
  IF p.priority > maxReady THEN
    maxReady := p.priority
  END;
END Enter;

PROCEDURE Select (VAR new: Process; priority: integer);
BEGIN
  LOOP
    IF maxReady < priority THEN new := nil; EXIT END;
    Get(ready[maxReady], new);
    IF(new # NIL) OR (maxReady = MinPriority) THEN
      EXIT
    END;
    maxReady := maxReady - 1
  END
END Select;
PROCEDURE CreateProcess (body: ADDRESS; priority: INTEGER; obj: OBJECT);
VAR p: Process;
BEGIN
  NEW(p); NewStack(p, body, obj);
  p.preempted := false;
  p.obj := obj; p.next := nil;
  RegisterFinalizer(p, FinalizeProcess);
  Acquire(Objects); (* module lock *)
  IF priority # 0 THEN p.priority := priority
  ELSE (* inherit priority of creator *)
      p.priority := running[ProcessorID()].priority
  END;
  Enter(p);
  Release(Objects)
END CreateProcess;
Stack Management

- Use virtual addressing
- Allocate stack in page units
- Use page fault for detecting stack overflow
- Deallocate stack via garbage collector (in process finalizer)

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateProcess</td>
<td>Allocate first frame</td>
</tr>
<tr>
<td>Page fault</td>
<td>Allocate another frame</td>
</tr>
<tr>
<td>Finalize</td>
<td>Deallocate all frames</td>
</tr>
</tbody>
</table>
Stack Allocation

Stack Initially

Stack After Extension
Context Switch

- Synchronous
  - Explicit
    - Terminate
    - Yield
  - Implicit
    - Awaiting condition
    - Mutual exclusion

- Asynchronous
  - Preemption
    - Priority handling
    - Timeslicing
PROCEDURE SwitchTo (VAR running: Process; new: Process);
Synchronous Context Switch (2)

Reduced process state

(cur) FP

running
new
retpc
dlink

(fp, sp)

(new) FP

running
new
retpc
dlink

(fp, sp)

fp
sp

Asynchronous Context Switch

Full process state

Registers saved by interrupt

(fp, sp)

(fp, sp)
Stack Layout after Interrupt

56 .SS
52 .ESP ; or haltcode

48 .EFLAGS
44 .CS
40 .EIP

36 .EBP ; err pushed by processor or glue code
32 .INT ; int number pushed by glue code

28 .EAX
24 .ECX
20 .EDX
16 .EBX
12 .ESP0
08 .ERR ; (was: EBP, exchanged by glue code)
04 .ESI
00 .EDI

state: State
First Level Interrupt Code (1)

PROCEDURE Interrupt;
CODE {SYSTEM.i386}
; called by interrupt handler (= glue code)
    PUSHAD ; save all registers (EBP = error code)

    ... ; now call all handlers for this interrupt

    POPAD ; now EBP = error code
    POP EBP ; now EBP = INT
    POP EBP ; now EBP = caller EBP
    IRETD
END Interrupt;
PROCEDURE Switch (VAR cur: Process; new: Process);
BEGIN
  cur.state.SP := SYSTEM.GETREG(SP);
  cur.state.FP := SYSTEM.GETREG(FP);
  cur := new;
  IF ~cur.preempted then (* return from call *)
    SYSTEM.PUTREG(SP, cur.state.SP);
    SYSTEM.PUTREG(FP, cur.state.FP)
    Release(Objects);
  ELSE (* return from interrupt *)
    cur.preempted := FALSE;
    SYSTEM.PUTREG(SP, cur.state.SP);
    PushState(cur.state.EFLAGS, cur.state.CS,
              cur.state.EIP, cur.state.EAX, cur.state.ECX,
              cur.state.EDX, cur.state.EBX, 0,
              cur.state.EBP, cur.state.ESI, cur.state.EDI);
    Release(Objects);
    JumpState POPAD
END Switch;
Example 1: Termination

PROCEDURE Terminate;
    VAR new: Process;
BEGIN
    Acquire(Objects);
    Select(new, MinPriority);
    Switch(running[ProcessorID()], new)
END Terminate;
Example 2: *Yield*

```plaintext
PROCEDURE Yield;
VAR id: INTEGER; new: Process;
BEGIN
   Acquire(Objects);
   id := ProcessorID();
   Select(new, running[id].priority);
   IF new ≠ NIL THEN
      Enter(running[id]);
      Switch(running[id], new)
   ELSE
      Release(Objects)
   END
END Yield;
```
Idle Activity in Objects

Idle = OBJECT
BEGIN{ ACTIVE, SAFE, PRIORITY(PrioIdle)}
LOOP
  REPEAT
    Machine.SpinHint
    UNTIL maxReady > MinPriority;
    Yield
  END
END Idle;
Example: Timeslicing

PROCEDURE Timeslice (VAR state: ProcessorState);
VAR id: integer; new: Process;
BEGIN
  Acquire(Objects);
  id := ProcessorID();
  IF running[id].priority # Idle THEN
    Select(new, running[id].priority);
    IF new # NIL THEN
      running[id].preempted := true;
      CopyState(state, running[id].state);
      Enter(running[id]);
      running[id] := new;
      IF new.preempted then
        new.preempted := false;
        CopyState(new.state, state)
      ELSE
        SwitchToState(new, state)
      END
    END
  END
  Release(Objects)
END Timeslice;
SwitchToState / CopyState

Prepare the state on the stack such that a return from interrupt will equal a return from Switch procedure

**PROCEDURE** SwitchToState(new: Process; VAR state: Machine.State);
BEGIN
state.SP := new.state.BP + AddressSize*2; (* effect of MOV ESP, EBP; POP EBP *)
SYSTEM.GET (new.state.BP, state.BP); (* effect of POP *)
SYSTEM.GET (new.state.BP + AddressSize, state.PC); (* effect of RET *)
END SwitchToState;

**PROCEDURE** CopyState* (CONST from: State; VAR to: State)
BEGIN
    to.EDI := from.EDI; to.ESI := from.ESI;
    to.EBX := from.EBX; to.EDX := from.EDX;
    to.ECX := from.ECX; to.EAX := from.EAX;
    to.BP := from.BP; to.PC := from.PC;
    to.CS := from.CS; to FLAGS := from.FLAGS; to.SP := from.SP
END CopyState;
Context Switching Scenarios

- From (running) to (synchronous suspended)
  - Call Switch

- From (synchronous suspended) to (synchronous suspended)
  - Exchange fp, sp

- From (synchronous suspended) to (asynchronous suspended)
  - Return from interrupt
  - Copy state

- From (asynchronous suspended) to (asynchronous suspended)
  - Copy state

- From (asynchronous suspended) to (running)
  - Return from interrupt
  - Copy state

- From (running) to (running)
  - Copy state
Synchronization

- Object locking
- Condition management
Object Descriptors

**ObjectHeader** = RECORD
    headerLock: BOOLEAN;
    lockedBy: Process;
    awaitingLock: ProcessQueue;
    awaitingCondition: ProcessQueue;

    ...

END;
Object Locking

PROCEDURE Lock (obj: object);
    VAR r, new: Process;
BEGIN
    r := running[ProcessorID()];
    AcquireObject(obj.hdr.headerLock);
    IF obj.hdr.lockedBy = nil THEN
        obj.hdr.lockedBy := r;
        ReleaseObject(obj.hdr.headerLock);
    ELSE
        Acquire(Objects);
        Put(obj.hdr.awaitingLock, r);
        ReleaseObject(obj.hdr.headerLock);
        Select(new, MinPriority);
        SwitchTo(running[ProcessorID()], new)
    END
END Lock;
Object Unlocking

PROCEDURE Unlock (obj: object);
VAR c: Process;
BEGIN
  c := FindCondition(obj.hdr.awaitingCondition)
  AcquireObject(obj.hdr.headerLock);
  IF c = NIL THEN
    Get(obj.hdr.awaitingLock, c);
  END;
  obj.hdr.lockedBy := c
  ReleaseObject(obj.hdr.headerLock);
  IF c # NIL THEN
    Acquire(Objects); Enter(c); Release(Objects)
  END;
END Unlock;
Condition Management

Condition Type

    TYPE
    Condition = PROCEDURE(fp: ADDRESS): BOOLEAN;

Condition Boxing

    PROCEDURE $Condition(fp: ADDRESS): BOOLEAN;
    BEGIN
        RETURN "condition from await statement"
    END $Condition;

Await Code

    IF ~$Condition(FP) THEN
        Await($Condition, FP, SELF)
    END
AWAIT Code

PROCEDURE Await (condition: Condition; fp: address; obj: object);
VAR r, t, new: Process;
BEGIN
   AcquireObject(obj.hdr.headerLock);
   c := FindCondition(obj.hdr.awaitingCondition);
   IF c = NIL THEN
      Get(obj.hdr.awaitingLock, c);
   END;
   obj.hdr.lockedBy := c
   Acquire(Objects);
   IF c # NIL THEN Enter(c) END;
   r := running[ProcessorID()];
   r.condition := condition;
   r.conditionFP := fp;
   Put(obj.hdr.awaitingCondition, r);
   ReleaseObject(obj.hdr.headerLock);
   Select(new, MinPriority);
   SwitchTo(running[ProcessorID()], new)
END Await;
PROCEDURE FindCondition (VAR q: ProcessQueue): Process;
VAR first, c: Process;
BEGIN
    Get(q, first);
    IF first.condition(first.conditionFP) THEN
        RETURN f
    END;
    Put(q, first);
    WHILE q.head ≠ first DO
      Get(q, c);
      IF c.condition(c.conditionFP) THEN
        RETURN c
      END;
      Put(q, c)
    END;
    RETURN NIL
END FindCondition;