

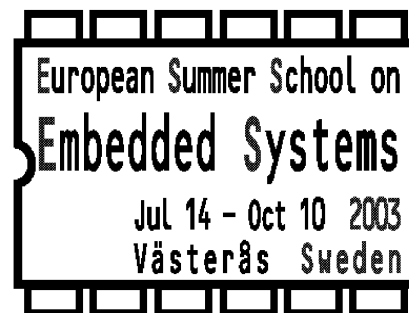
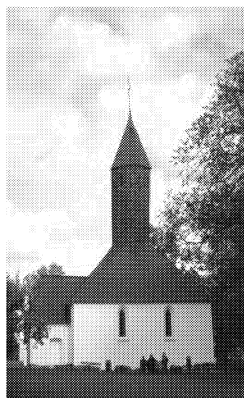
MÄLARDALENS HÖGSKOLA

ESSES 2003

European Summer School on Embedded Systems

Lecture Notes Part XVI

Embedded Systems: Operating System and Middleware for Embedded Systems



Editors: Ylva Boivie, Hans Hansson, Jane Kim, Sang Lyul Min

Västerås, September 8-12, 2003

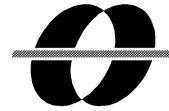
ISSN 1404-3041

ISRN MDH-MRTC-110/2003-1-SE

MRTC

MÄLARDALEN REAL-TIME
RESEARCH CENTRE

www.mrtc.mdh.se



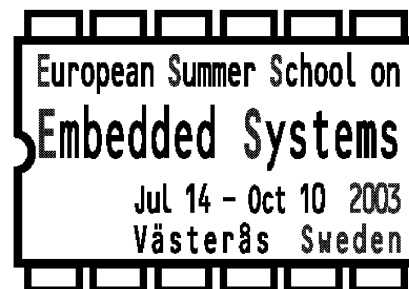
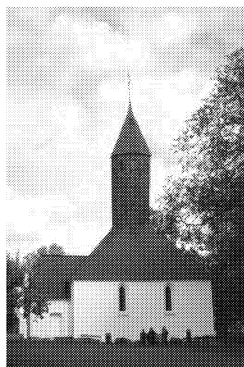
MÄLARDALENS HÖGSKOLA

ESSES 2003

European Summer School on Embedded Systems

Lecture Notes Part XVII

Embedded Systems: Operating System and Middleware for Embedded Systems



Editors: Ylva Boivie, Hans Hansson, Jane Kim, Sang Lyul Min

Västerås, September 8-12, 2003

ISSN 1404-3041

ISRN MDH-MRTC-110/2003-1-SE

MRTC

MÄLARDALEN REAL-TIME
RESEARCH CENTRE

www.mrtc.mdh.se

Rob van Ommering

Philips National Lab

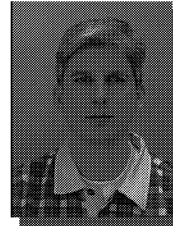
The Koala Component Model

Overview

European Summer School on Embedded Systems

Västerås, Sverige
September 2003

Rob van Ommering
Philips Research
Eindhoven, The Netherlands
Rob.van.Ommering@philips.com



Personal Data

- ◆ '58 - '64 at home 
- ◆ '64 - '70 Primary School Geldrop 
- ◆ '70 - '76 Gymnasium Augustinianum Eindhoven 
- ◆ '76 - '82 Technical University Eindhoven
Technical Physics 
- ◆ '82 - '88 Philips Research Eindhoven
Robotics, Computer Vision, Machine Learning 
- ◆ '88 - '94 Philips Centre for Software Technology Eindhoven
Industrial Application of Formal Specifications
- ◆ '94 - '00 Philips Research Eindhoven
SW Architecture Visualization and Verification
Software Architecture of Consumer Products 



1. The Problem
2. Components
3. Koala
4. Architecture
5. A Pattern
6. Product Line



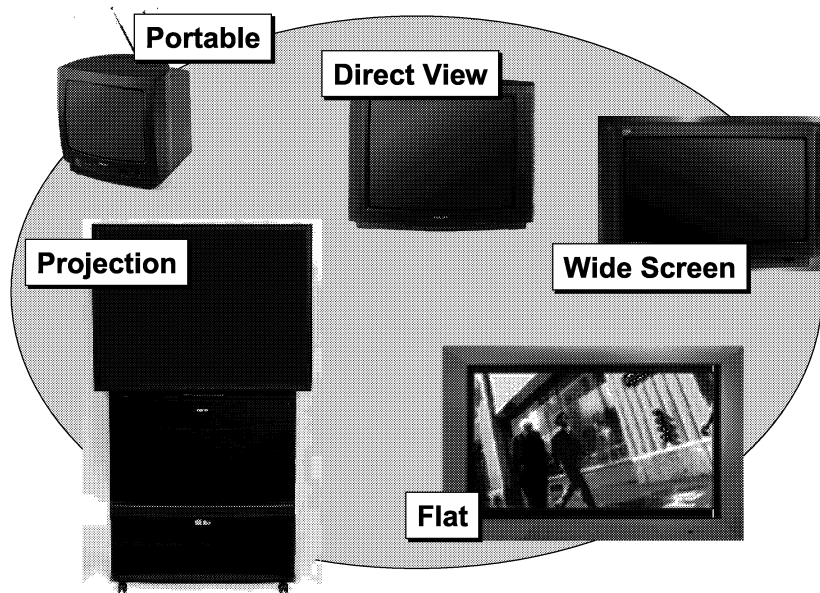
Part I

The Problem



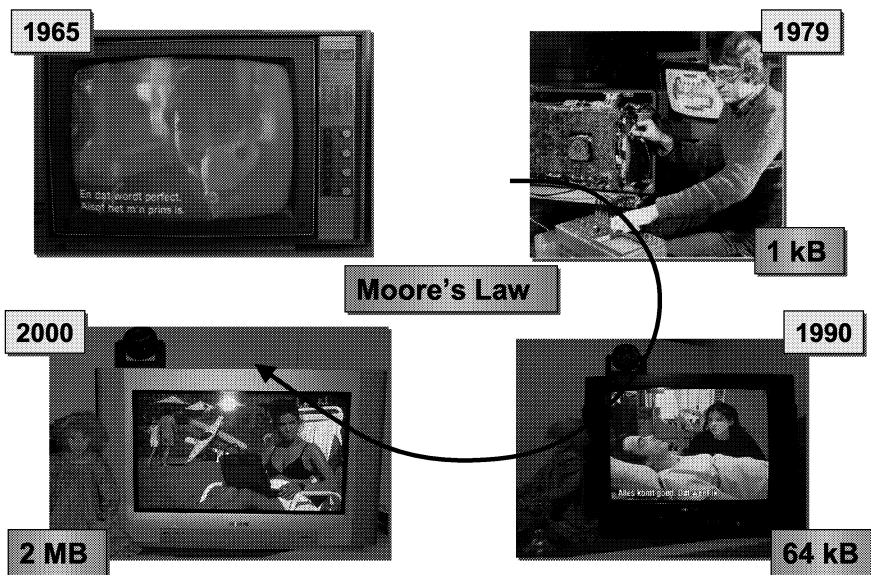
The TV Domain

5

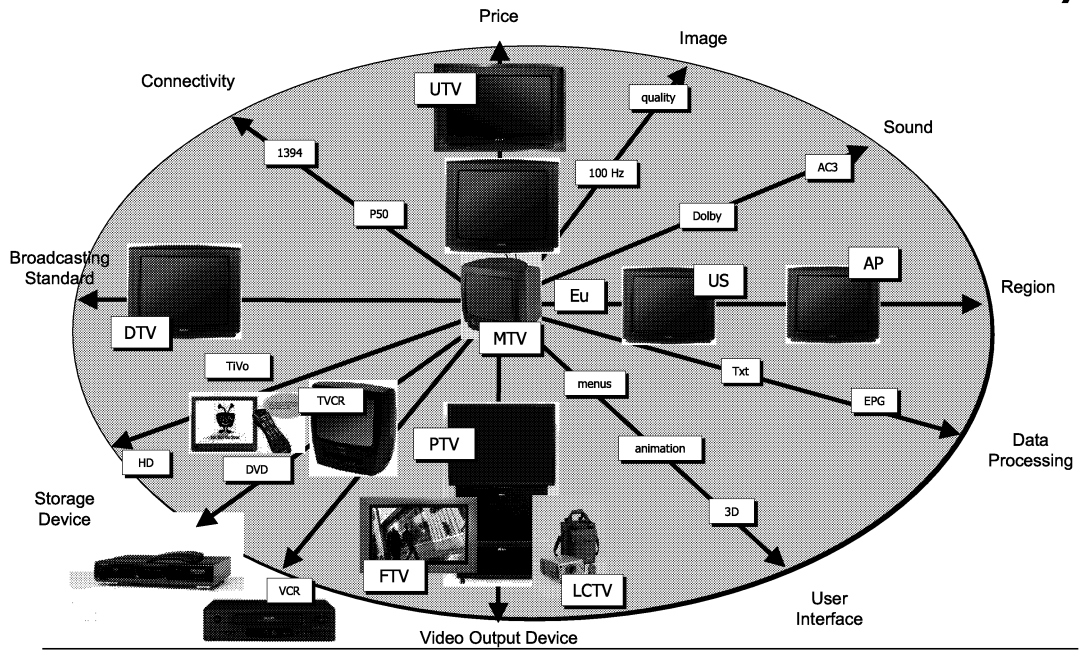


A Brief History of TV...

6



TV Diversity



© 2003 Koninklijke Philips Electronics NV

The Koala Component Model, Overview, September 2003, RvO



PHILIPS

There's more than just TV...



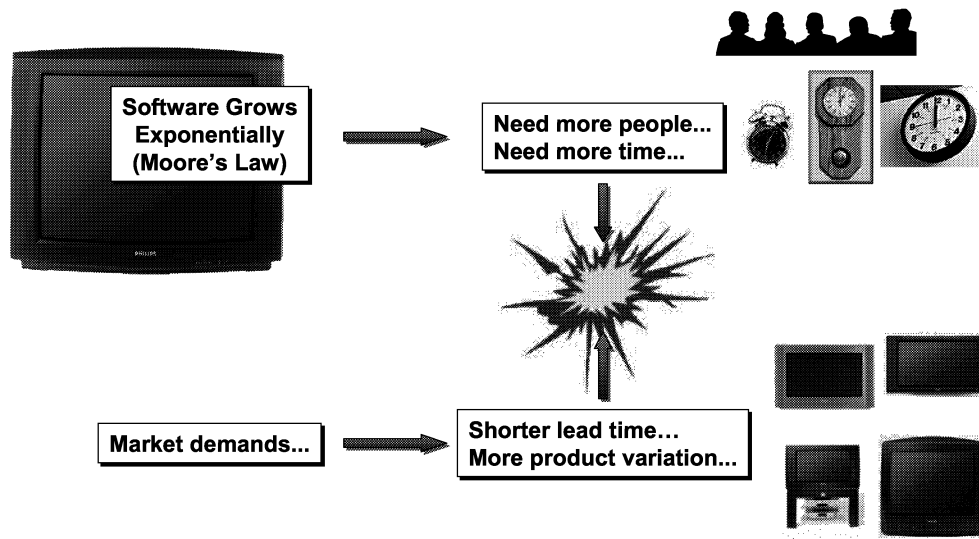
© 2003 Koninklijke Philips Electronics NV

The Koala Component Model, Overview, September 2003, RvO

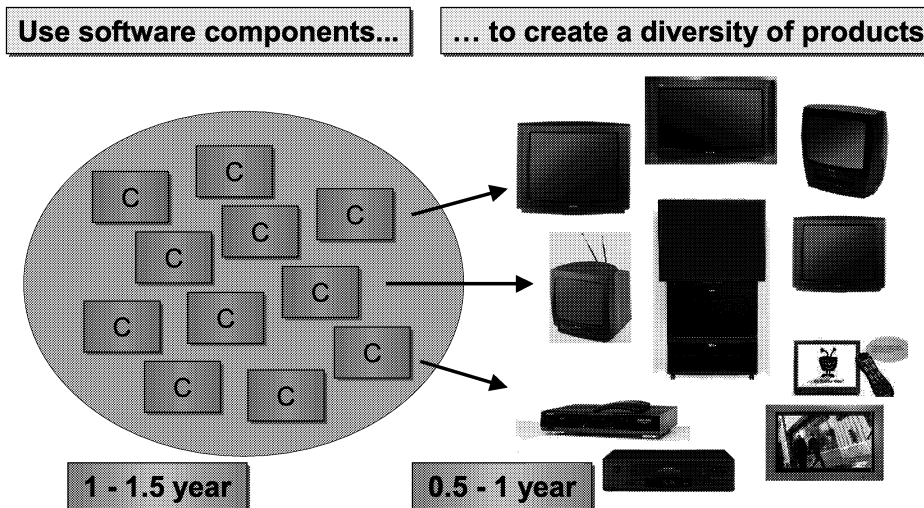


PHILIPS

Problem Statement

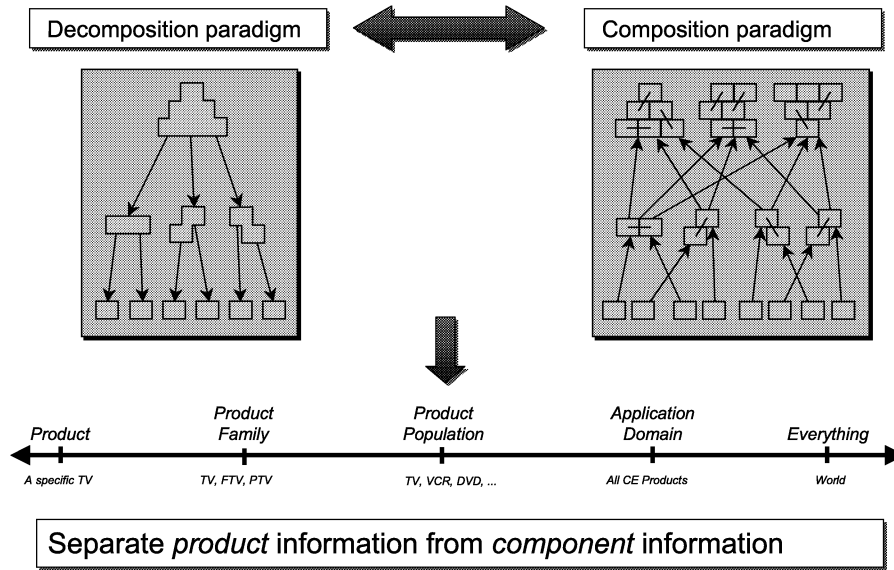


Solution: Use Components



A Paradigm Shift

11



12

Part II

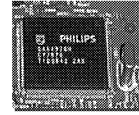
Components



What is a Software Component?

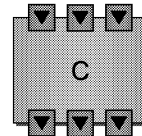
A software component is:

- a unit of composition with
- contractually specified interfaces and
- explicit context dependencies only



A software component can be:

- deployed independently and is
- subject to composition by third parties

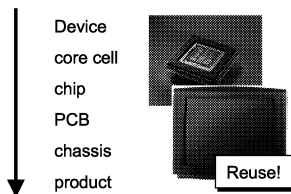


Clemens Szyperski



Looking for a Component Model...

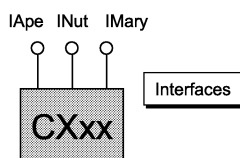
(1) Hardware



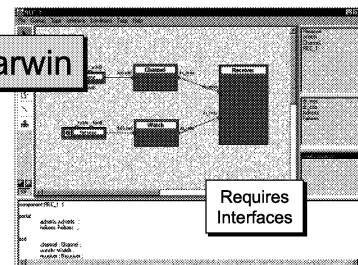
(3) Visual Basic



(2) Microsoft's COM



(4) Darwin



(5) JavaBeans

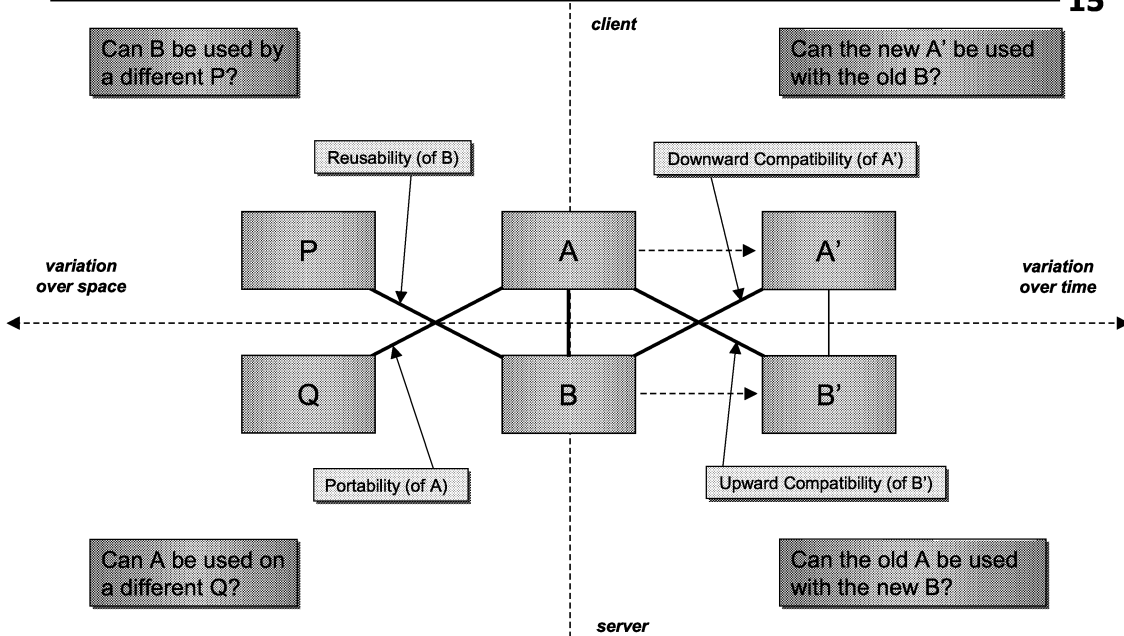
(6) Corba

(7) ...



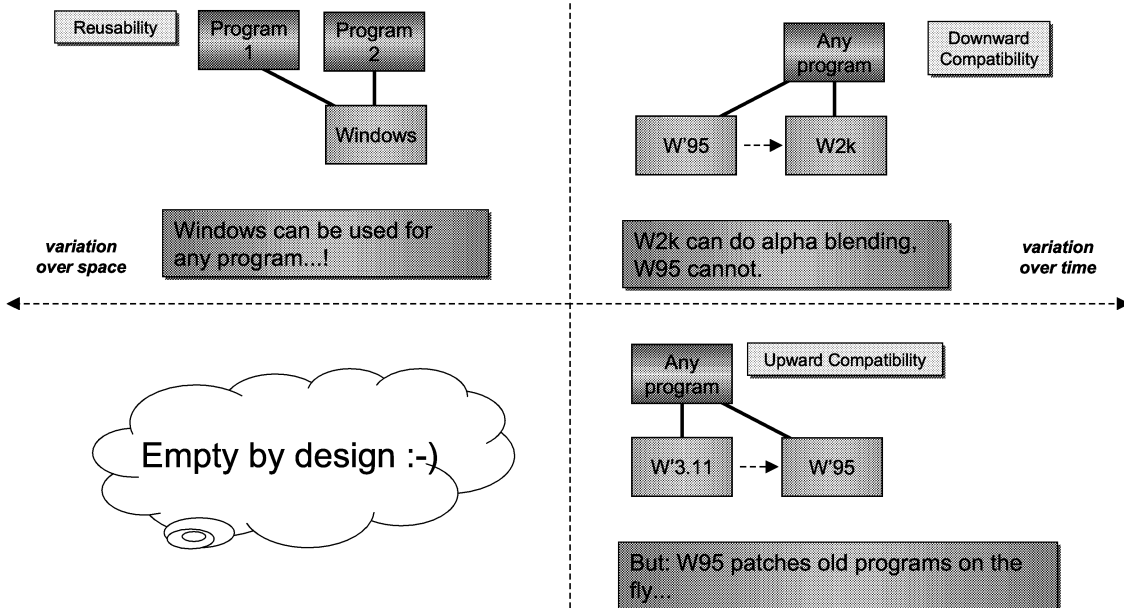
Independent Deployment - The 4 Issues

15

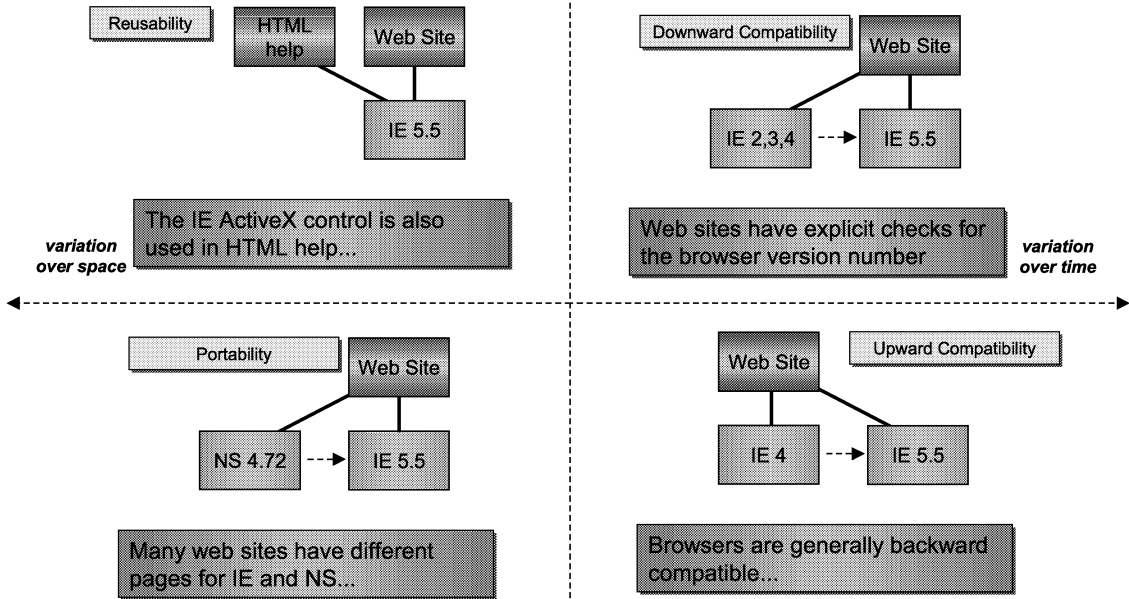


Examples in Windows

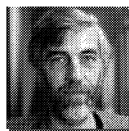
16



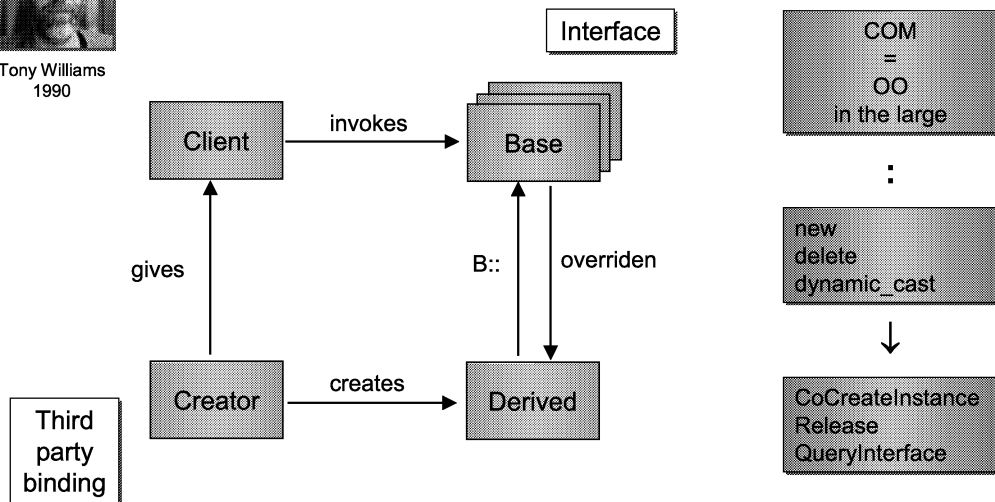
Browser Examples



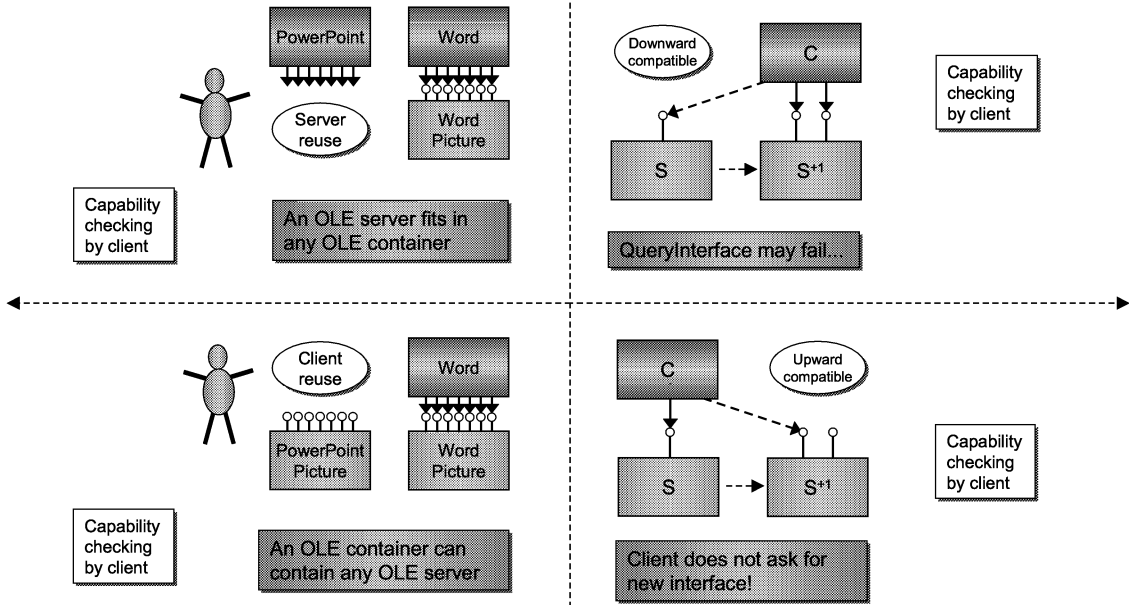
Using Inheritance...



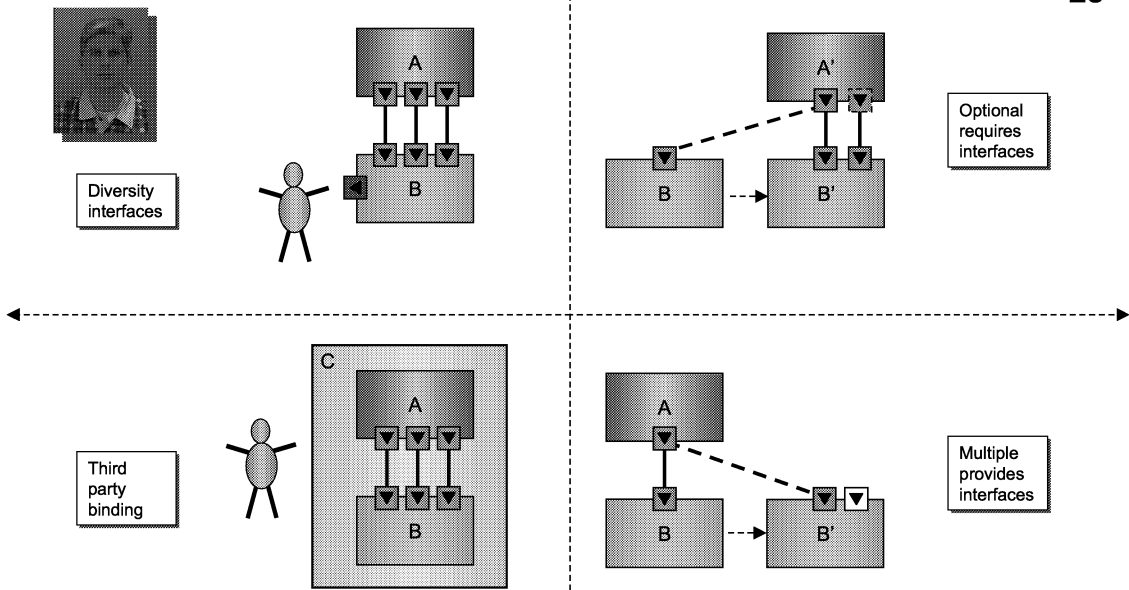
Tony Williams
1990



COM and OLE



Koala...



Part III

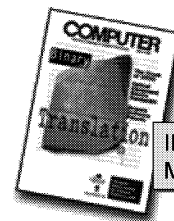
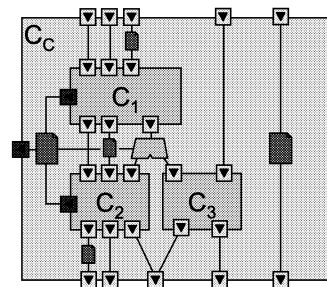
Koala



The Koala Component Model


Koala offers:

- a. Provides interfaces and interfaces as first class citizens
- b. Requires interfaces and 3rd party binding
- c. Aggregation and Gluing
- d. Parameterization, optional interfaces and 'dynamic' binding

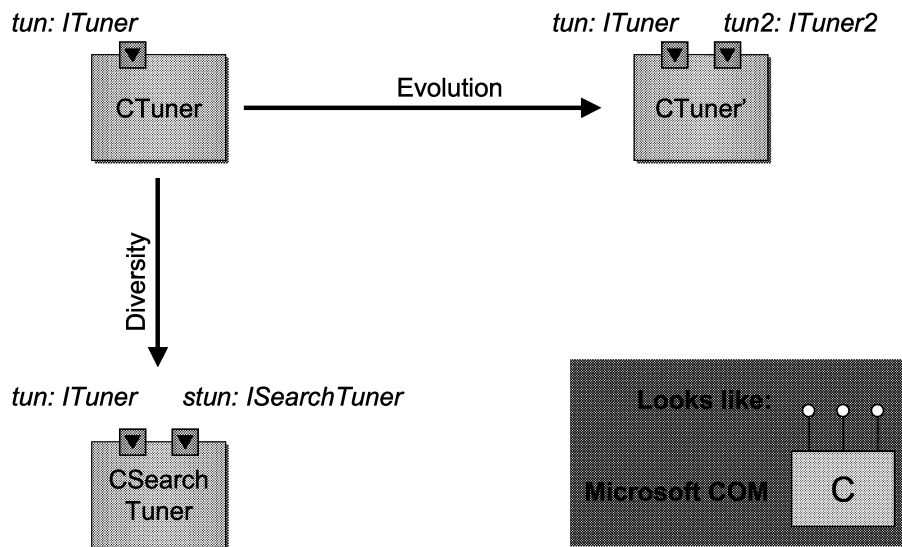


IEEE Computer
March 2000



Provides Interfaces

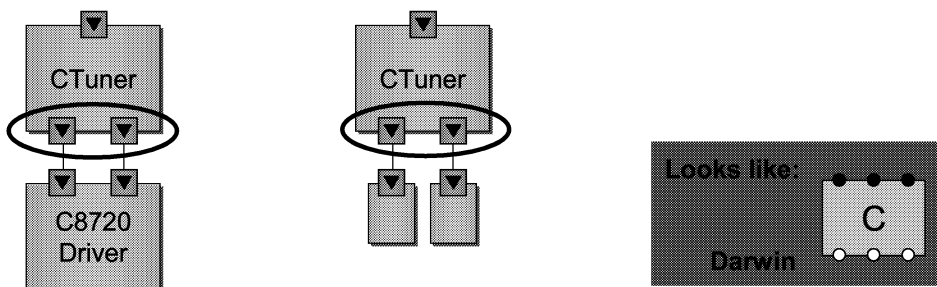
23



Requires Interfaces

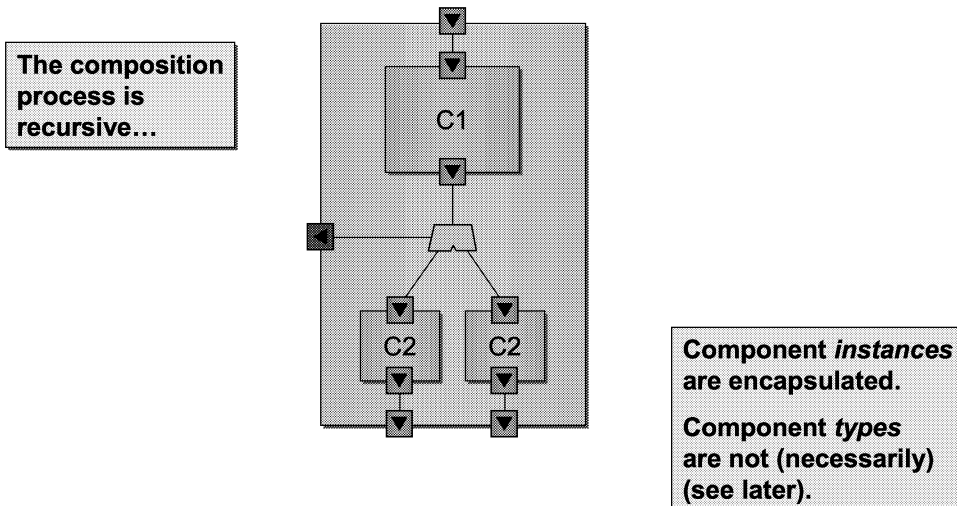
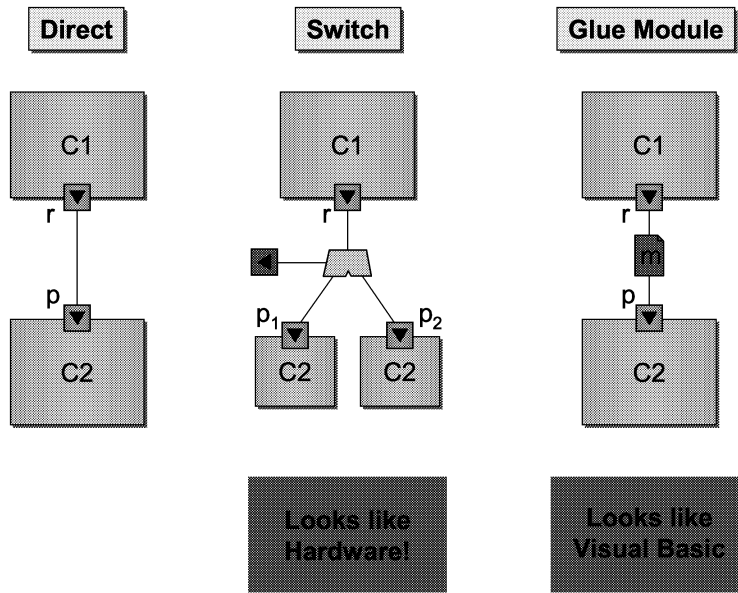
24

All context dependencies are made explicit...
...and are bindable by a third party

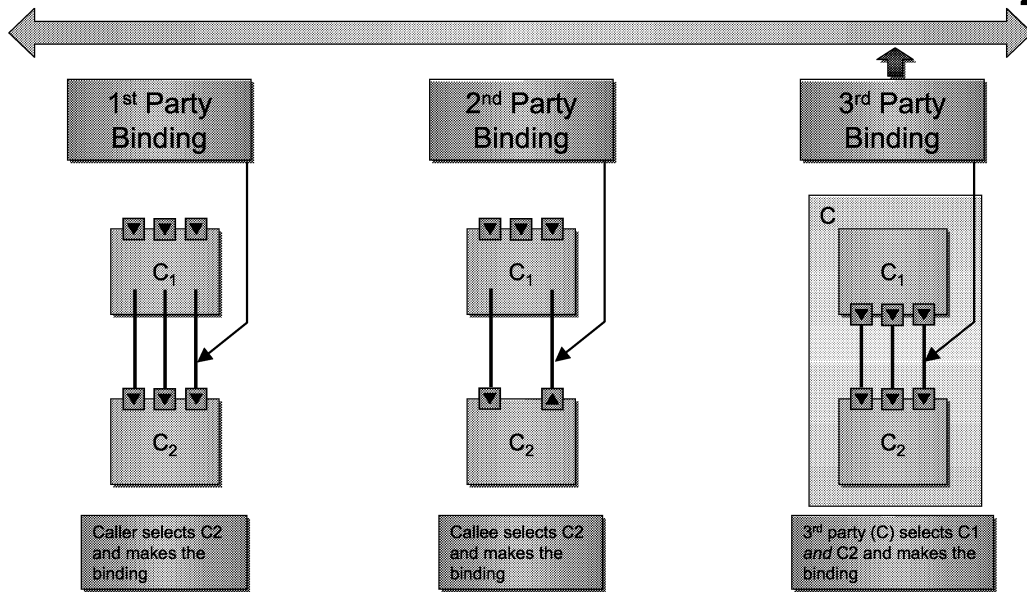


...so they can be bound differently in another product

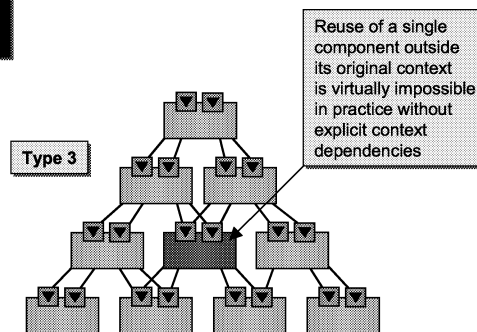
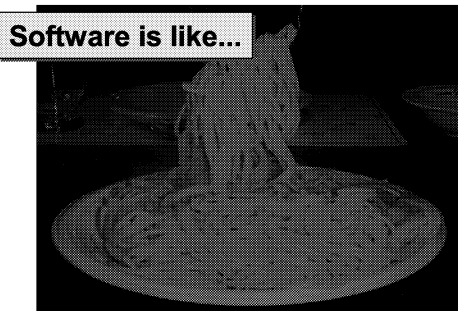




Third Party Binding



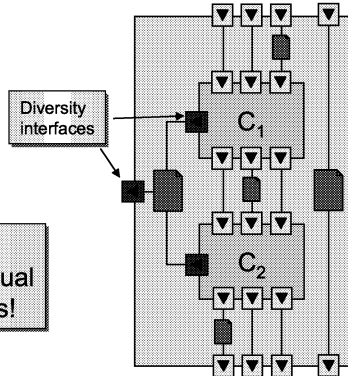
Requires Interfaces and Binding



Diversity Interfaces

Components can only be made **usable and reusable** if they are **parameterized** over a (large number of) items.

Compare diversity interfaces with Visual Basic property lists!



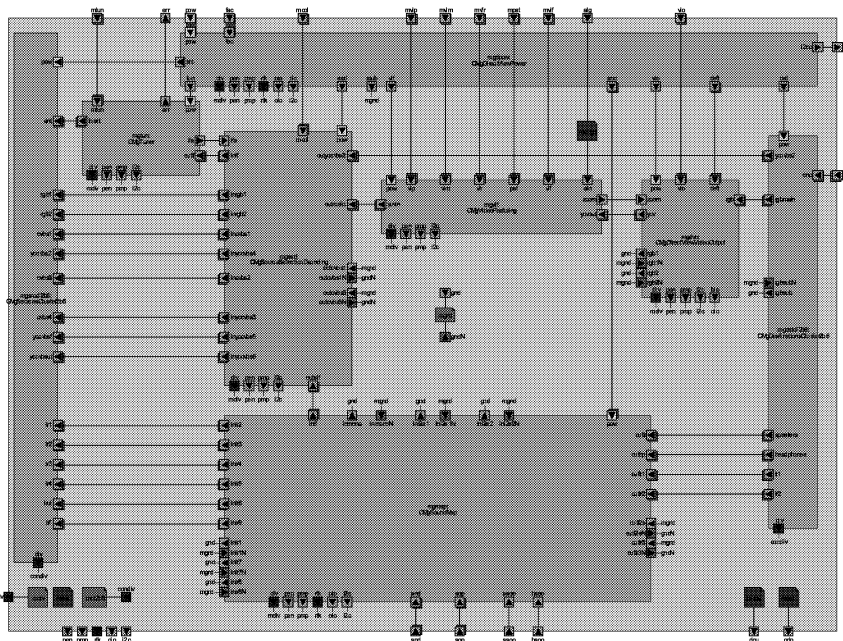
Parameters (functions) in diversity interfaces can be glued with modules.



This allows to express inner diversity in terms of outer diversity!

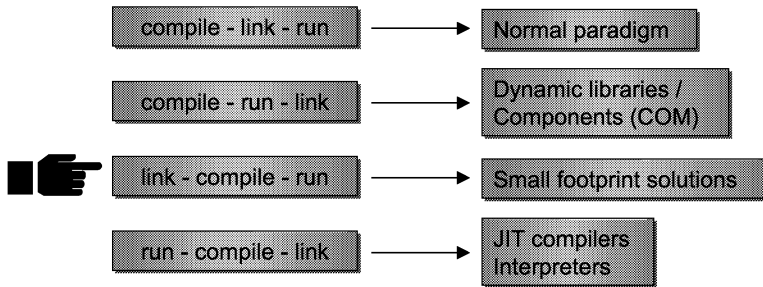


Koala - Example



Koala separates the model from the implementation

The current implementation minimizes footprint by utilizing *late compile-time binding* :

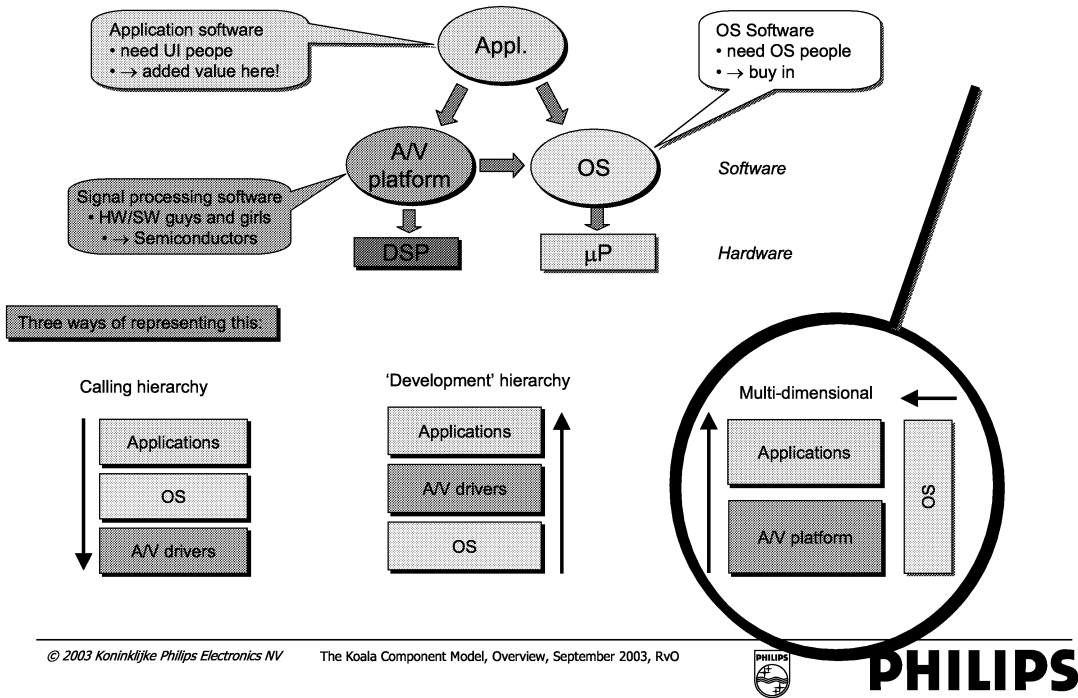


Part IV

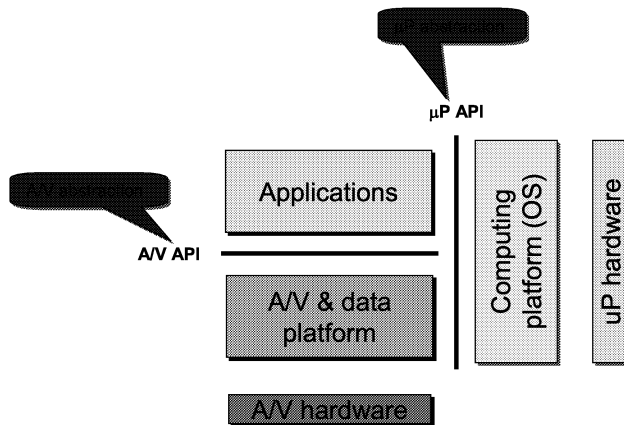
Architecture



Architecture: Layers

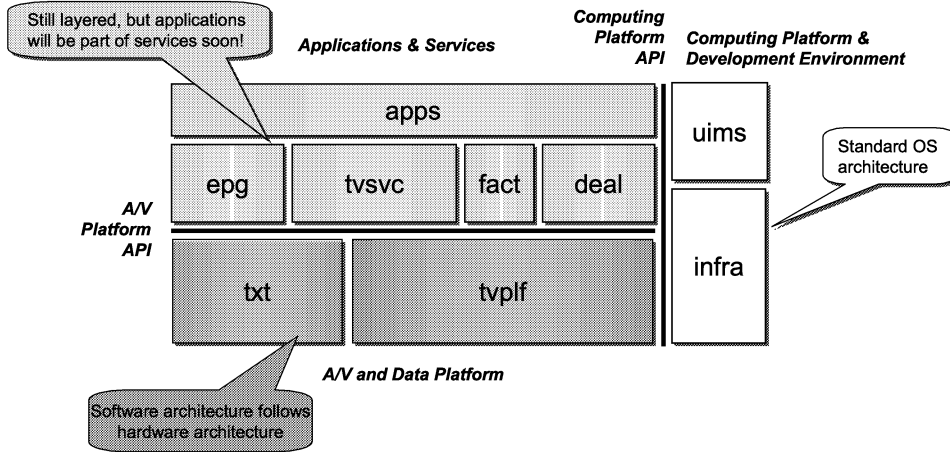


Architecture: APIs



Architecture: Subsystems

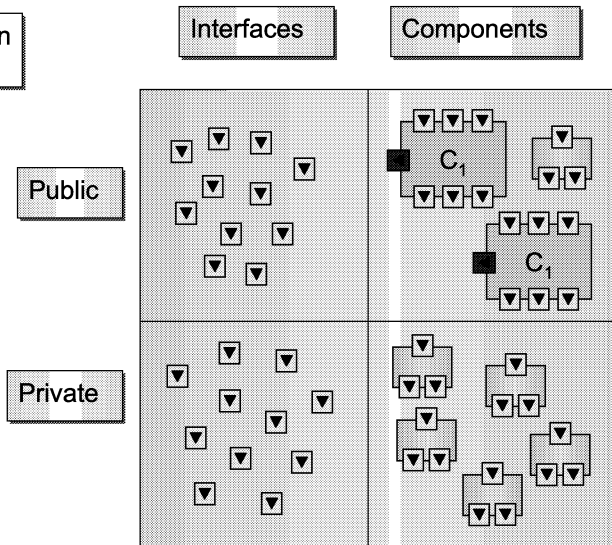
The following subsystems are currently defined:



Architecture: Packages

To streamline this, the notion of **package** is needed.

A **package** is a collection of private and public component and interface definitions

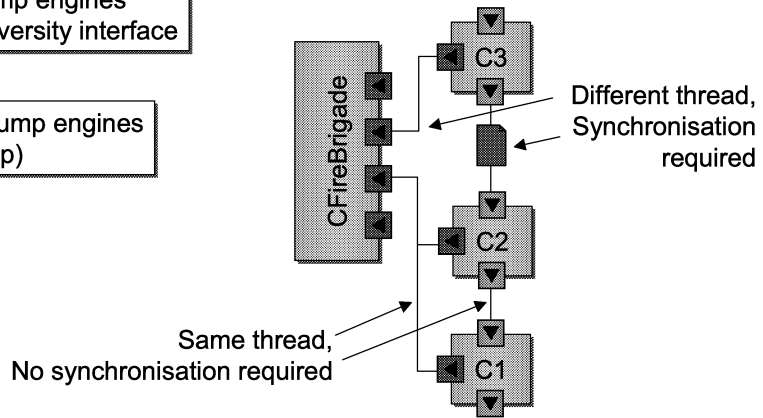


Multi Threading

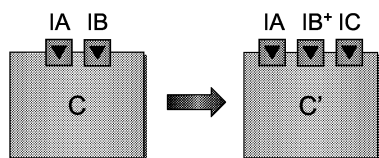
Problem: many (>100) activities but few (<10) threads

Step 1: use message pumps created on virtual pump engines required through a diversity interface

Step 2: bind these to pump engines (a real dispatcher loop)



Evolution

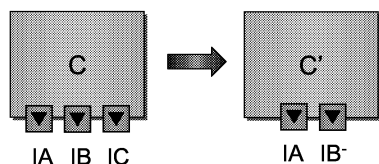


Provide more...

$$IB^+ \subseteq IB$$

$$C' \subseteq C$$

Koala subtypes interfaces based on set inclusion of functions



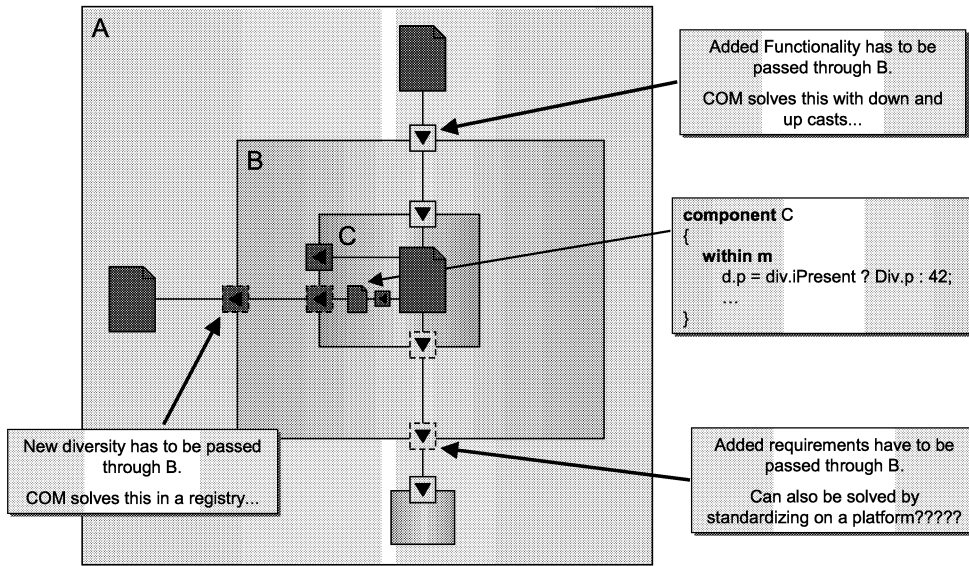
Require less...???

$$IB^- \supseteq IB$$

$$C' \subseteq C$$

Koala reports an error if a non-existing interface is bound...!





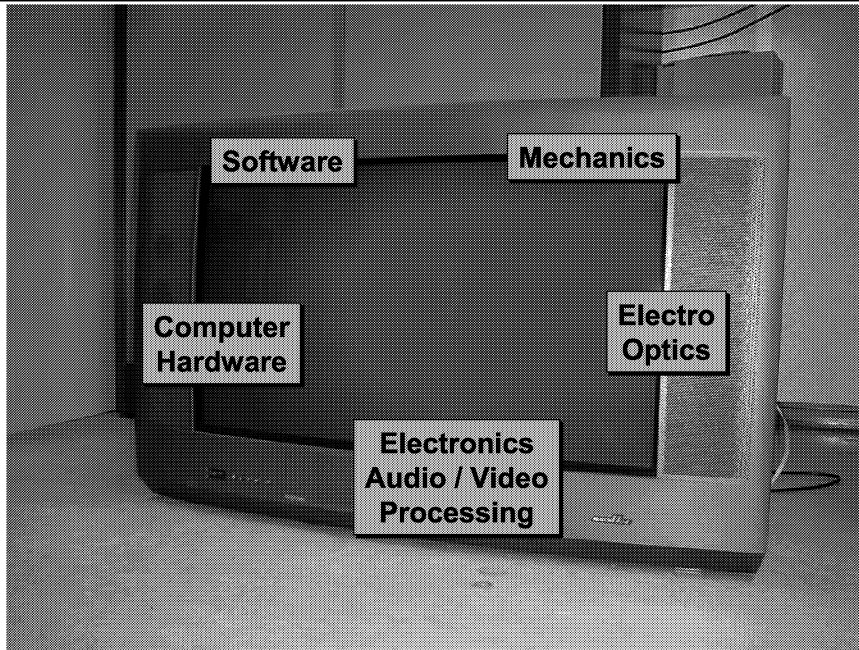
Part V

A Pattern



What's in a TV?

41



© 2003 Koninklijke Philips Electronics NV

The Koala Component Model, Overview, September 2003, RvO



PHILIPS

The Inside of a TV

42



© 2003 Koninklijke Philips Electronics NV

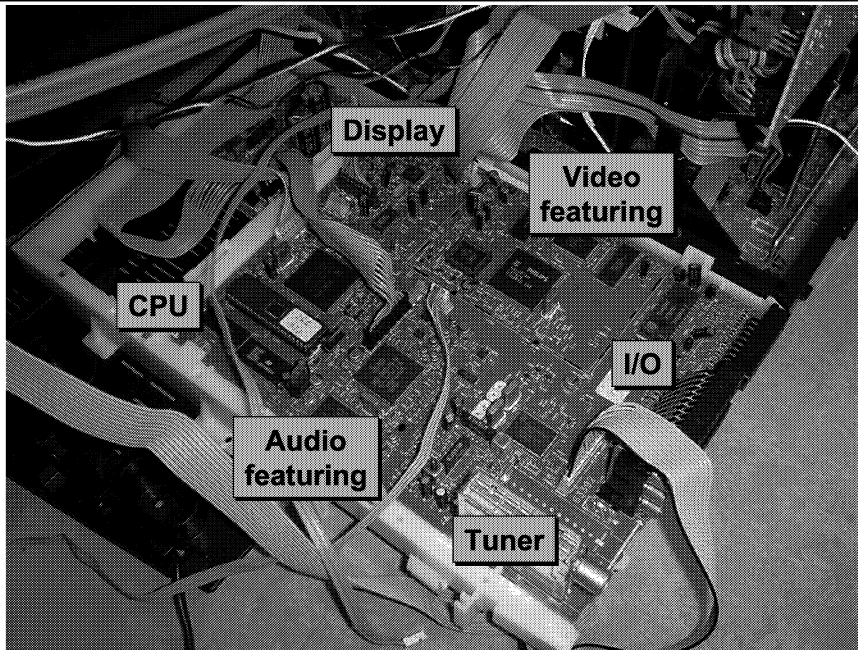
The Koala Component Model, Overview, September 2003, RvO



PHILIPS

The Small Signal Panel

43



© 2003 Koninklijke Philips Electronics NV

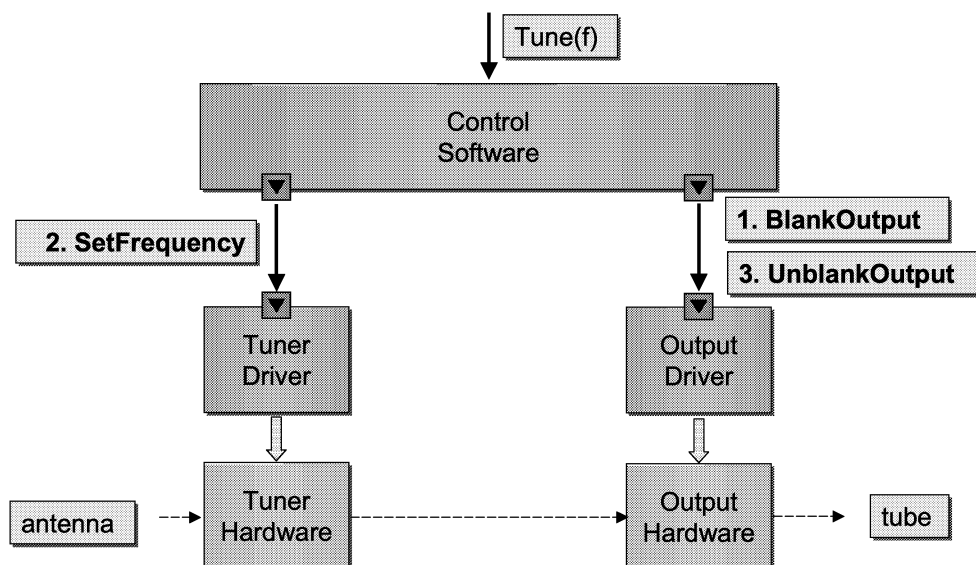
The Koala Component Model, Overview, September 2003, RvO



PHILIPS

Example Design Problem

44



© 2003 Koninklijke Philips Electronics NV

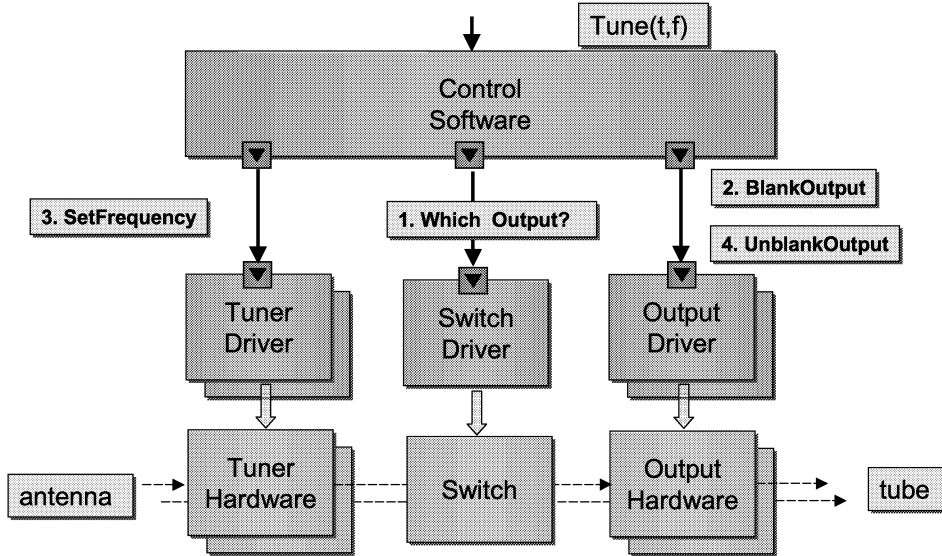
The Koala Component Model, Overview, September 2003, RvO



PHILIPS

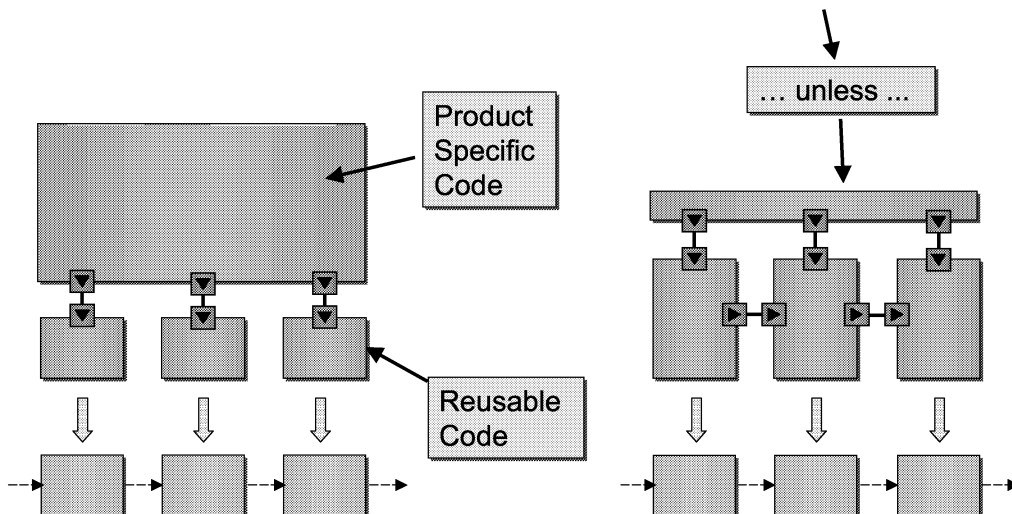
Example Design Problem (2)

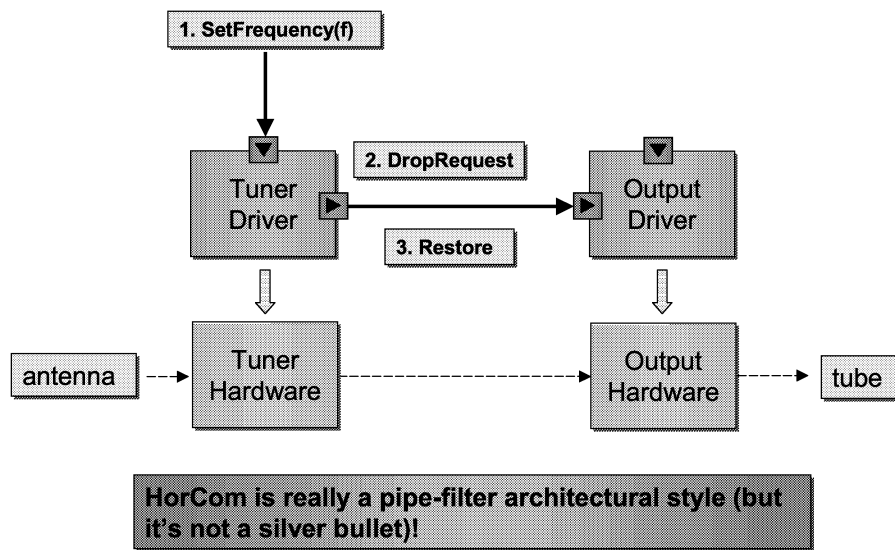
45



Control software is difficult to decompose...

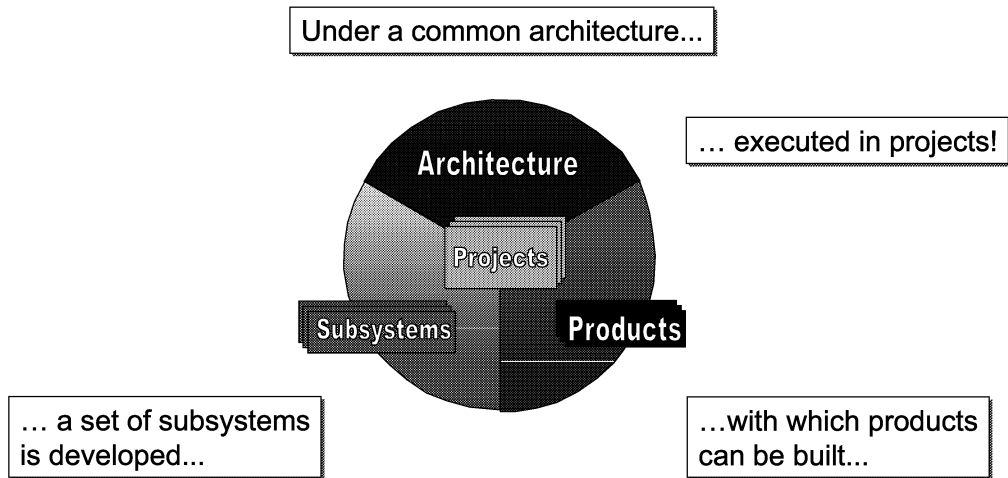
46



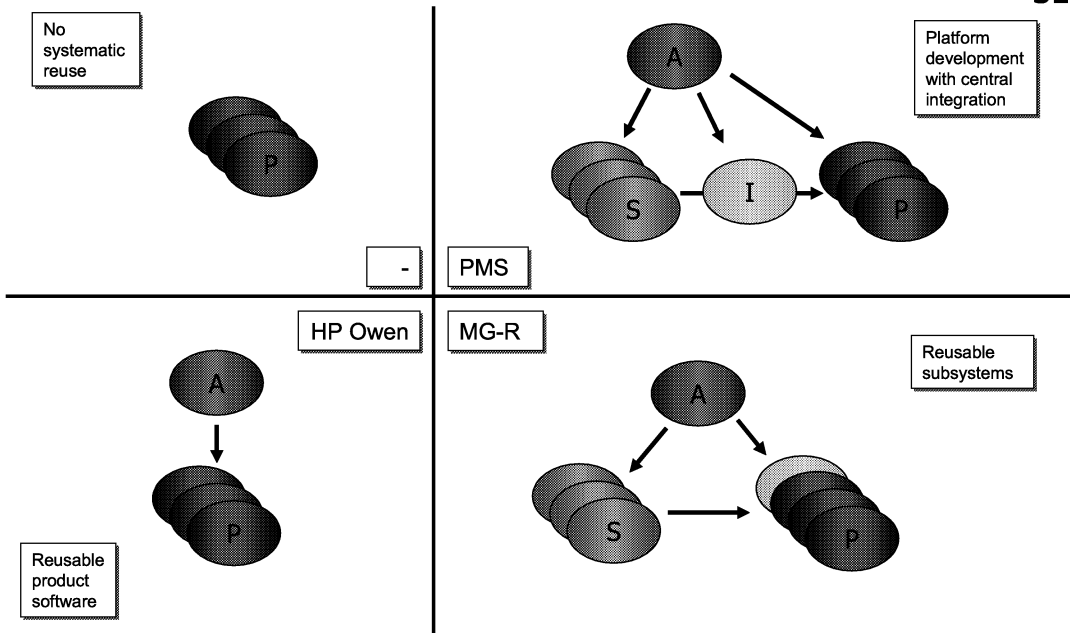


Part VI

Product Line

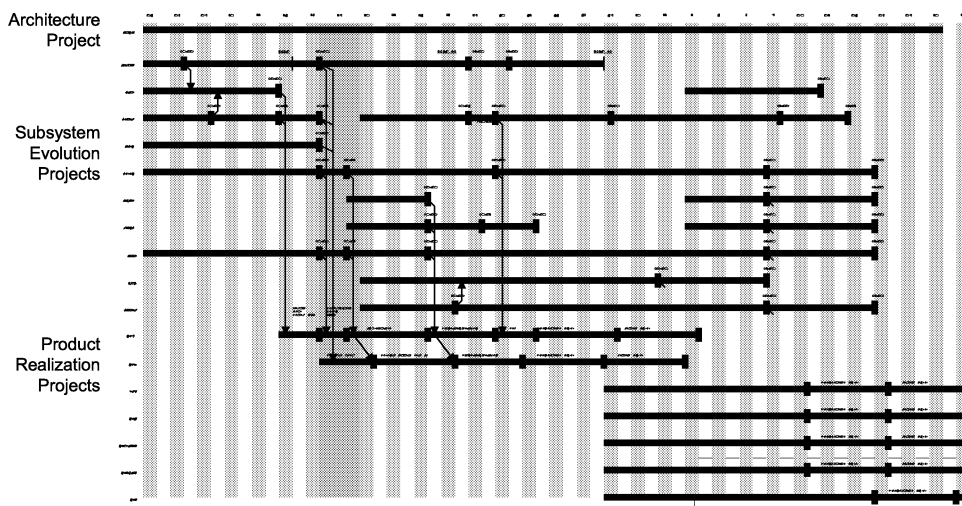


Reuse Approaches Compared



Roadmapping

Product and subsystem releases are carefully planned:





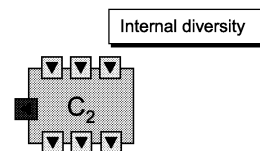
Separate component information from interface information
 Separate component external behaviour from component implementation
 Use *does* instead of *shall*.



Configuration Management

Distinguish between:

- ◆ *versions*
 - ◆ *temporary variants*
 - ◆ *permanent variants*
- of components.

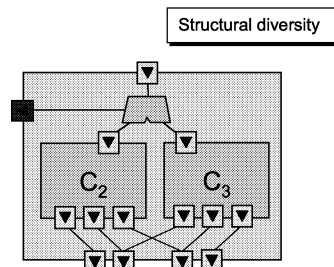


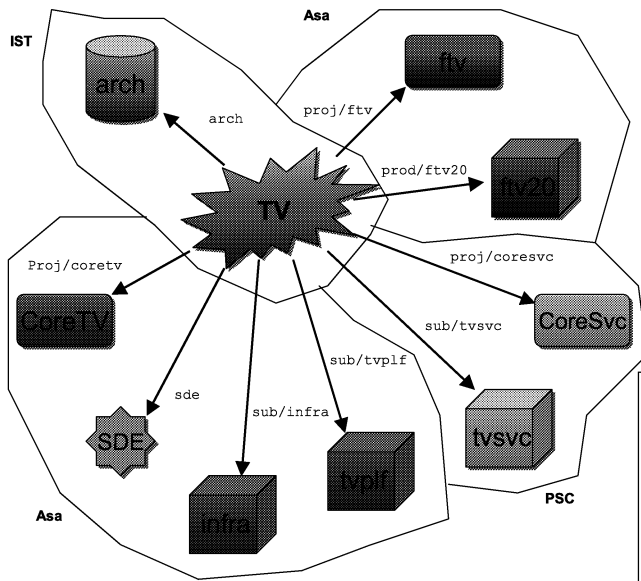
We use our CM system for:

- ◆ *versions*
- ◆ *temporary variants*

But we use the component model for:

- ◆ *permanent variants*





Every project, subsystem and product has its own web site. These sites can be reached from the home page, ... but they are physically distributed over the world

- Formats:**
- ◆ **project:** own house style
 - ◆ **platform:** use directory structure
 - ◆ **product:** use directory structure
 - ◆ **architecture:** free
 - ◆ **SDE:** free
- Update daily!!!**



The End



The Koala Component Model

Rob van Ommering
Maarten Pennings

September 2003

Contents

1. Concepts

2. The Basic Model

3. More on Interfaces

4. Function Binding

5. Constants

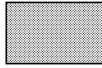
6. Optional Interfaces

7. Switches

Koala supports the following concepts:

- why component model
- components
- interfaces
- modules
- binding

Koala's Concepts



A *component* is a unit of encapsulation.



An *interface* is a small and coherent set of functions.



A *module* is a unit of code.

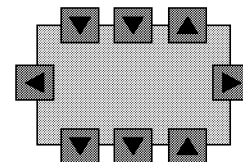


A *binding* is a connection between interfaces.

Components

A component is a piece of software that is non trivial in size (an asset to the company), but that does *not* contain any *configuration specific* information.

A component *provides* and *requires* interfaces, and can interact through these interfaces only.



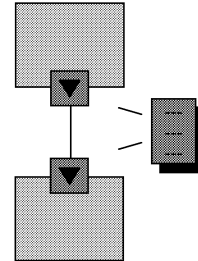
Interfaces

We treat interfaces as *first class citizens*.

- interfaces are *unidirectional*;
- each interface has a *definition*;

Interface definitions are *units of reuse*.

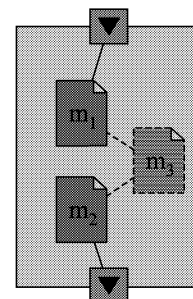
- an interface definition is always *shared* between a ‘provider’ and a ‘requirer’;
- *variants* of components may provide or require the same interfaces.



Modules

For Koala, a module is a *hand-written C* file and a *generated* header file.

- not all modules in a component need to be declared to Koala;
- header files for undeclared modules must be hand written (of course);
- communication between modules within a component is completely free;
- communication with the environment must go through interfaces.

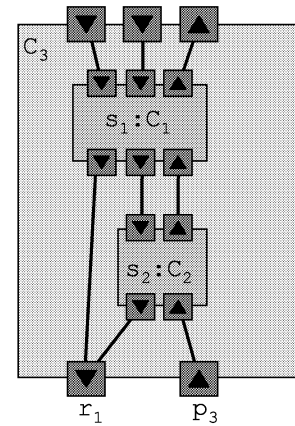


Compound Components

A group of components can be seen as a component again...

This allows us to create *reusable subsystems* (or standard designs).

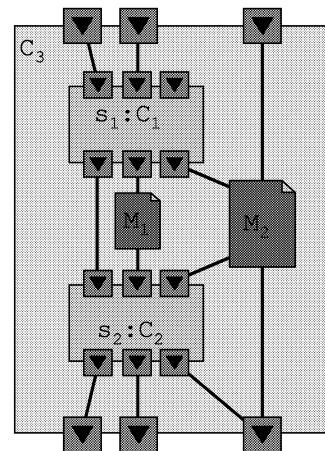
A subcomponent in a compound component is a *copy* (instance) of a reusable component (type).



Configurations

Definitions:

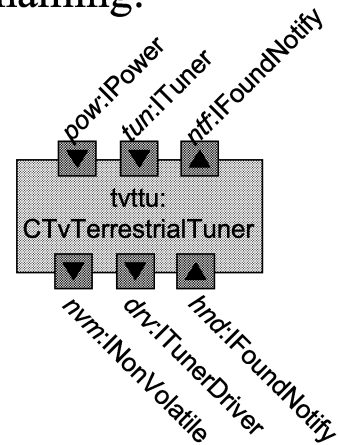
- a *basic component* is a component without subcomponents;
- a *compound component* is a component with one or more subcomponents;
- a *configuration* is a component without provides and requires interfaces, a top-level component.



Before we proceed, let's discuss naming:

- a component has a globally unique long name and short name (aka prefix)
- each interface has a local (or instance) name unique to the component
- each interface is associated with an interface definition

Note that two or more interfaces in a single component *may* have the same definition!



1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

Koala supports three 'languages':

- DD: data type definitions
- ID: interface definitions
- CD: component definitions

Languages

- **CD** - Component definition
describes components referring to component definitions and interface definitions
- **ID** - Interface definition
describes function prototypes and constants referring to datatype definitions
- **DD** - Datatype definition
describes datatypes referring to other datatypes

DD

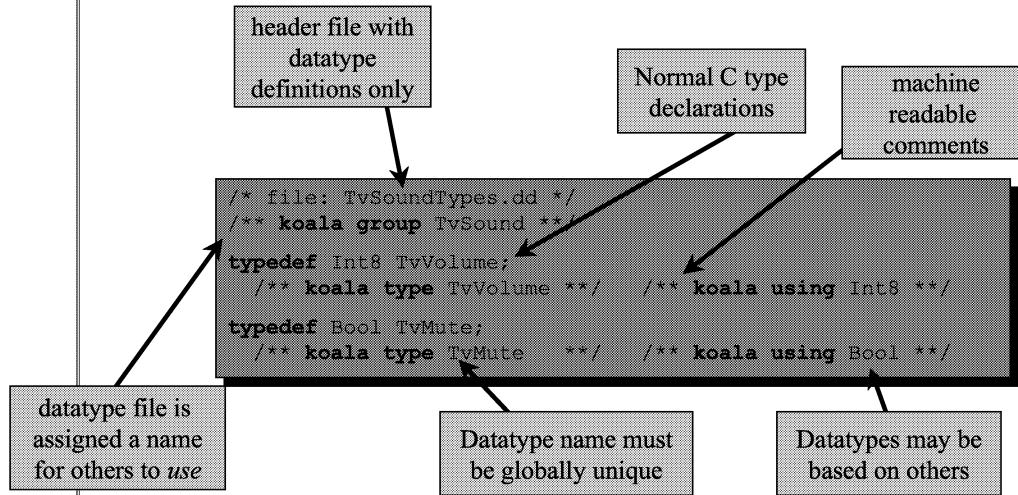
An *datatype definition* declares a datatype that is used for typing parameters in functions (occurring in interfaces)

Ordinary C header file with *machine readable comments* embedded



```
typedef char Int8; /** koala type Int8 **/  
typedef char Bool; /** koala type Bool **/
```

DD - rules



DD – advise

Avoid data types on your interfaces because they

- have *global* scope (so at least prefix them)
- can't *evolve* (win32 struct trick)
- can't be *copied* into variant package

But

- ones from C are ok (*int, char, void*)
- ones from *infra* are ok (*Bool, Nat8, Int32, ...*)

An *interface definition* describes the syntax and semantics of a small set of functions:

Syntax: names and types of functions and arguments.

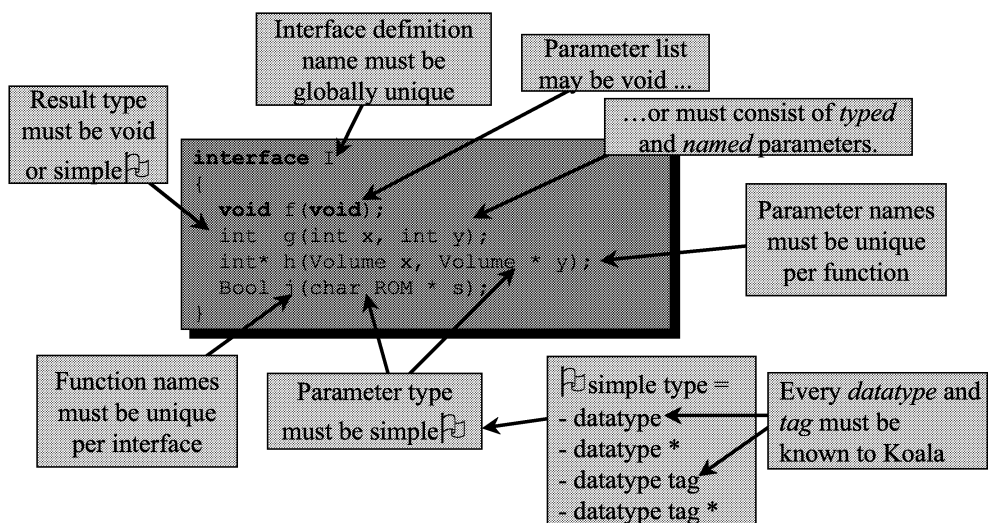
Semantics: a (simple) logical model of the behaviour.

Interfaces consist of *functions* (in a broad sense – see later).

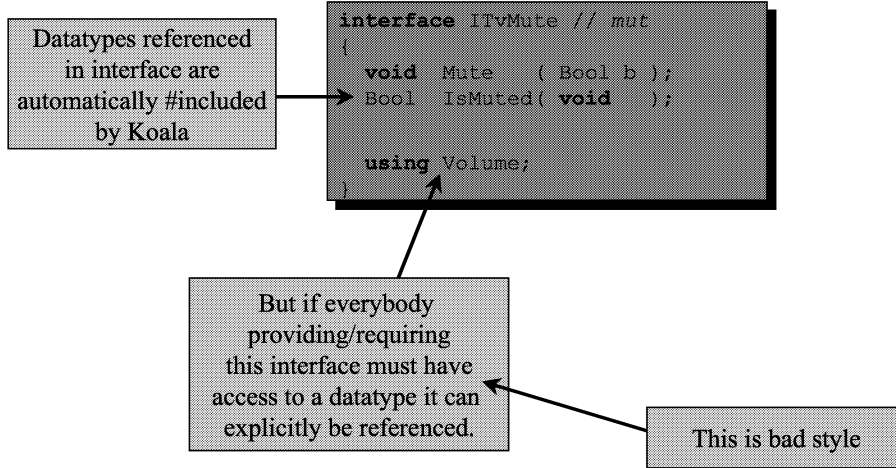
ID

```
interface ITvSoundControl // sdc
{
    void SetVolume(TvVolume v);
    void SetTreble(TvTreble b);
}
```

*



ID - rules



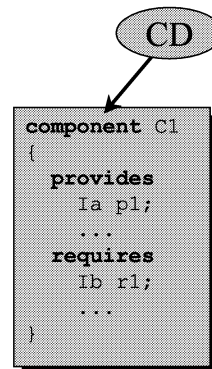
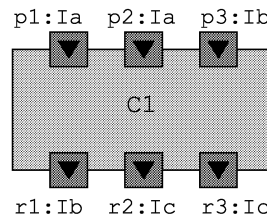
CD

In a *component definition*, the interfaces provided and required by the component can be declared.

A component is implemented as a set of C and H files.

It is mandatory to have a directory per component.

Part of the internal structure is also described to Koala.

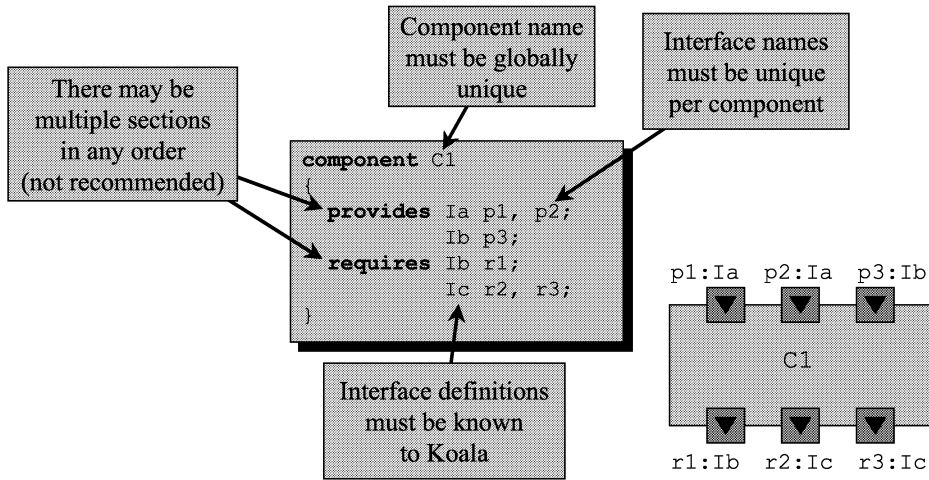


*

Koala

Basic

CD - rules



Koala, Sep 2003, RvO, 19

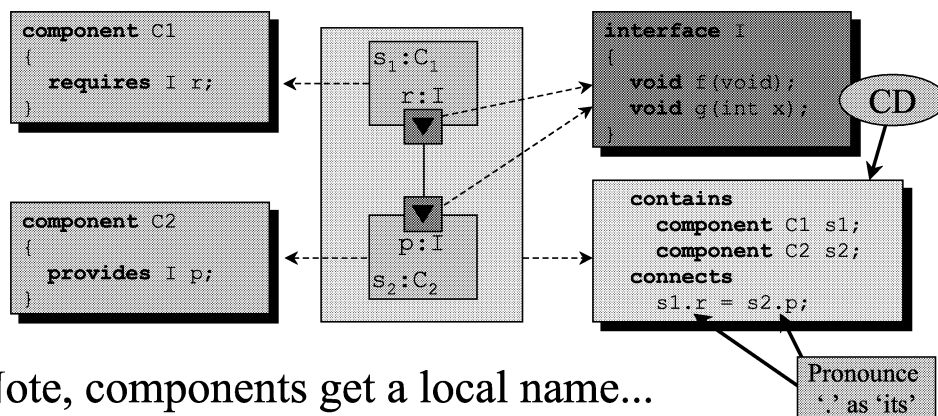
Koala Workshop

Koala

Basic

Binding

Interfaces must be bound *tip to base*:



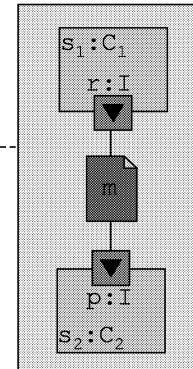
Koala, Sep 2003, RvO, 20

Koala Workshop

Code may be inserted in a binding in the form of a *module*...

- the header file of *m* is generated;
- the C file of *m* is hand written.

```
contains
component C1 s1;
component C2 s2;
module m;
connects
s1.r = m;
m     = s2.p;
```



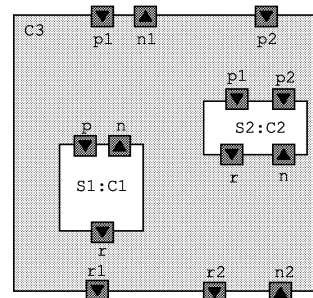
This allows to *fine tune* components at the level of configurations.

An interface is a *contract*. So, if a component provides an interface, it must somehow implement it.

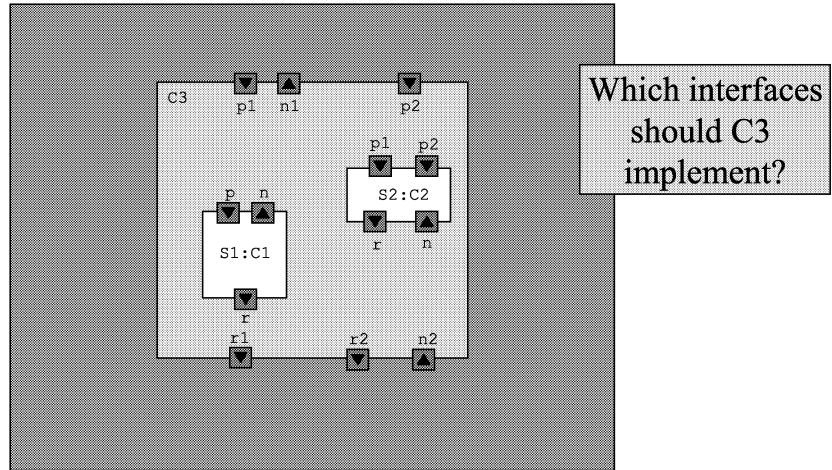
A component should implement

- its provided interfaces
- the required interfaces of its subcomponents

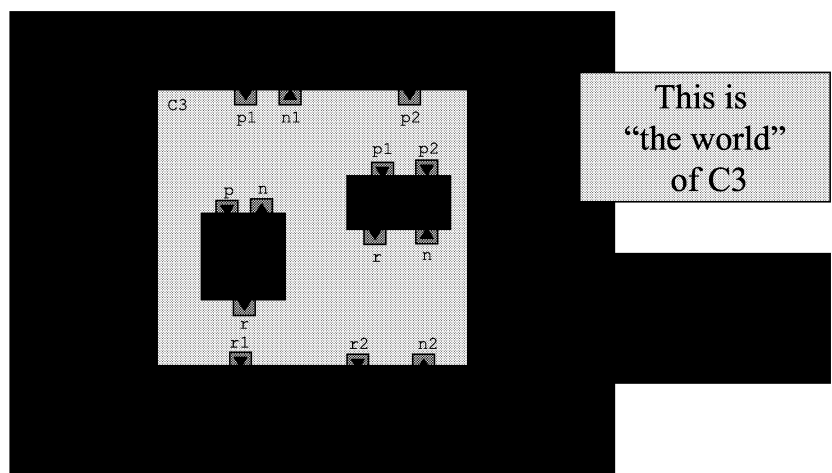
To formalize this, we have introduced the notion of *tips* and *bases*.

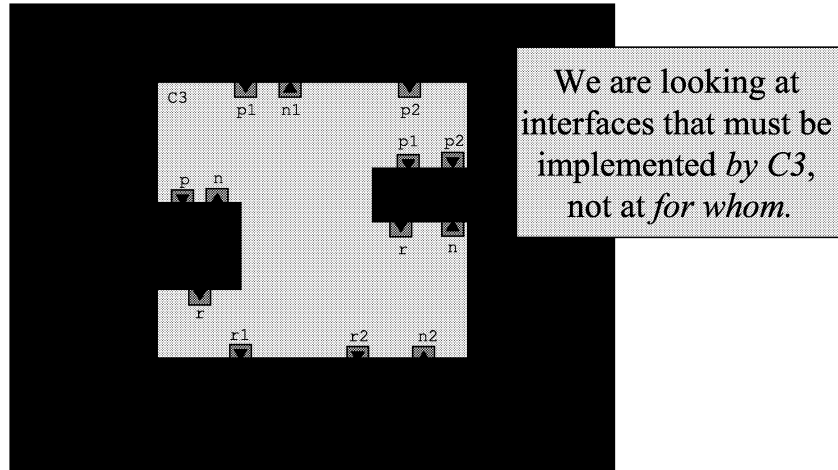


Tips and bases - 1



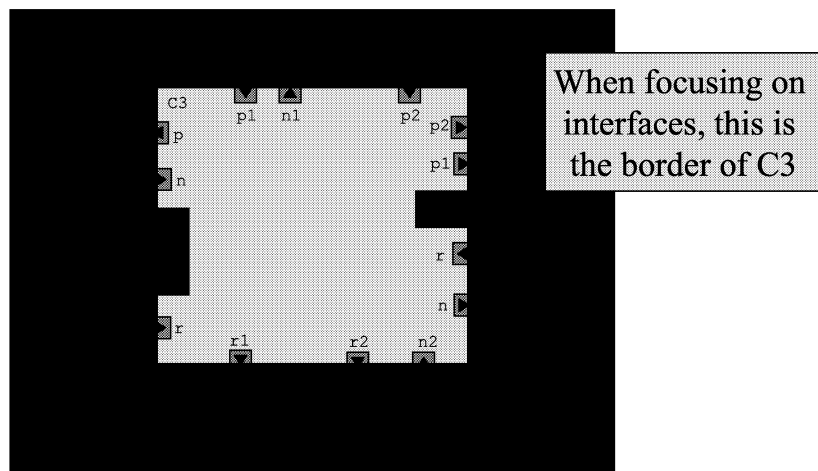
Tips and bases - 2





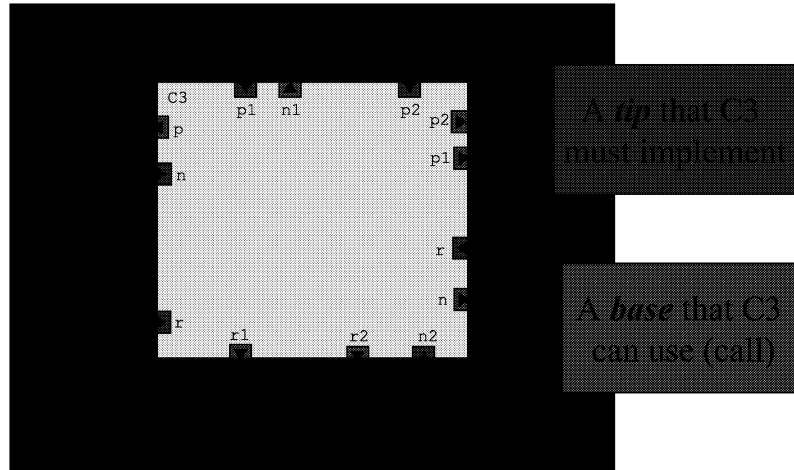
Koala, Sep 2003, RvO, 25

Koala Workshop



Koala, Sep 2003, RvO, 26

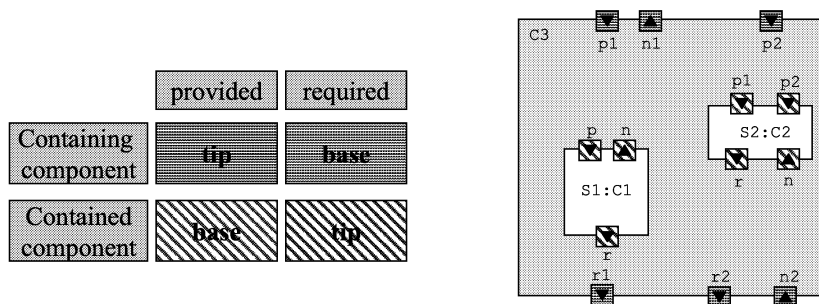
Koala Workshop



Koala, Sep 2003, RvO, 27

Koala Workshop

The containing component must provide code for all *tips* and it can use all *bases*.



Koala, Sep 2003, RvO, 28

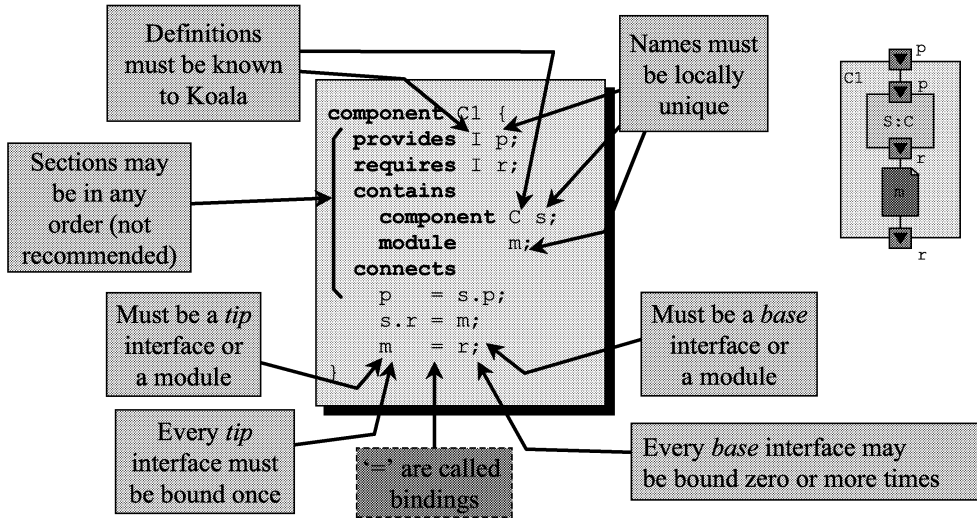
Koala Workshop

*

Koala

CD - rules

Basic



Koala, Sep 2003, RvO, 29

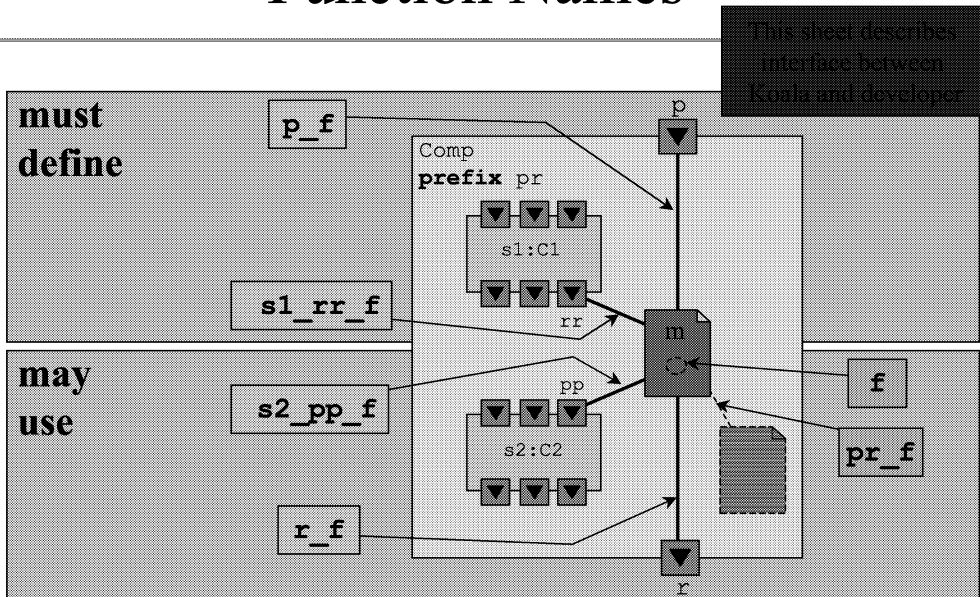
Koala Workshop

*

Koala

Function Names

Basic



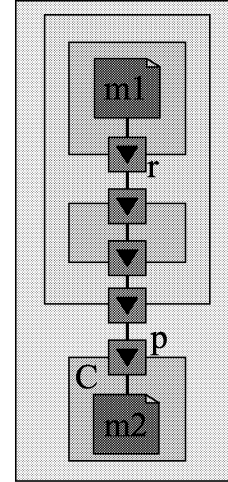
Koala, Sep 2003, RvO, 30

Koala Workshop

Example

```
/* Koala generated: _m1.h */
#define r_Func C__p_Func
extern void C__p_Func(int x);
```

```
/* file: m1.c */
#include "_m1.h"
void SomeFunc(void)
{
    r_Func(54);
}
```



No runtime
nor size
penalty!

Koala generated

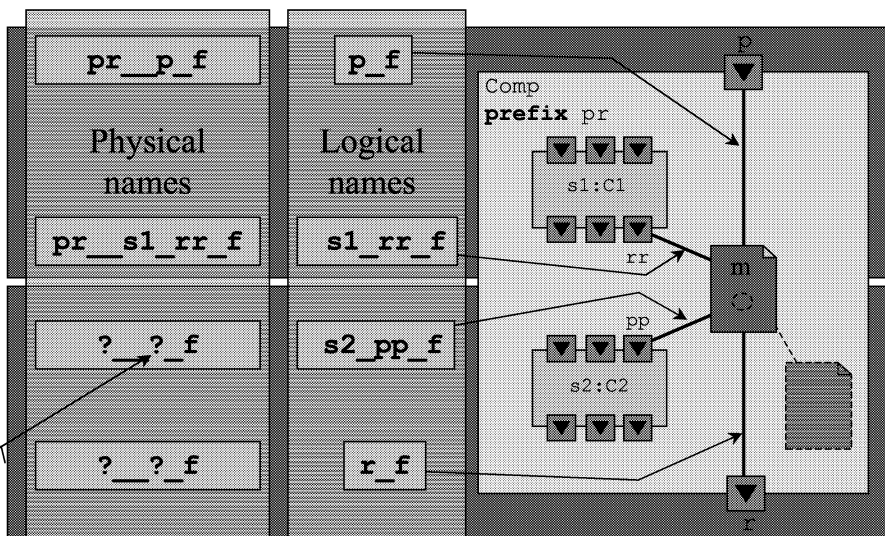
handcrafted

```
/* Koala generated: _m2.h */
#define p_Func C__p_Func
extern void C__p_Func(int x);
```

```
/* file: m2.c */
#include "_m2.h"
void p_Func(int x)
{
    ... x ... ;
}
```

*

Name mangling by binding



This is
not pp!

*

Koala

Naming

Basic

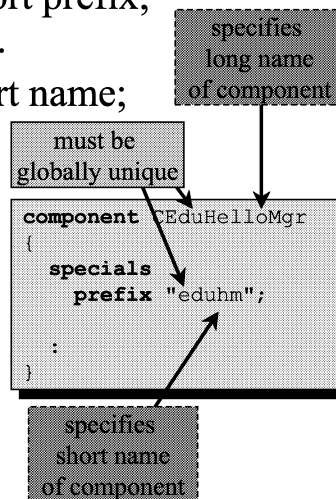
A package has a so-called long and short prefix;
e.g. *Edu* and *edu* for course package.

Every component has a long and a short name;
e.g. *CEduHelloMgr* and *eduhm*.

Note that

- the *component long name* starts with the *package long prefix*;
- the *component short name* starts with the *package short prefix*;

Both names are specified in the cd file.



Koala, Sep 2003, RvO, 33

Koala Workshop

*

Koala

CD - rules


Basic

```
component C1 {
  :
  uses
  Bool, Pump;
  :
}
```

If you need access to a datatype that is not available through referenced interfaces

Koala, Sep 2003, RvO, 34

Koala Workshop



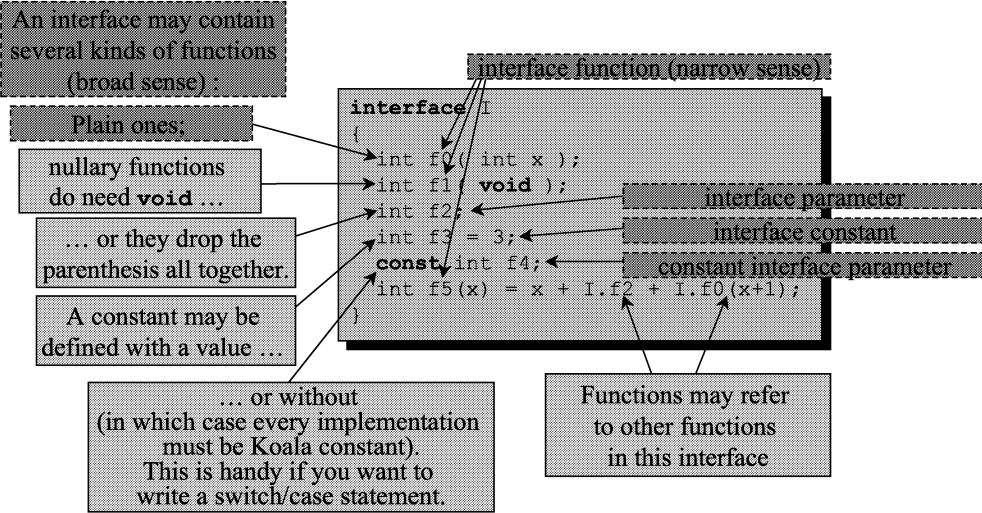
Hello, MG-R!

1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

We discuss

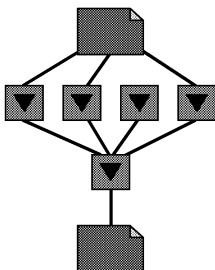
- functions in broad sense
- private interfaces
- interface compatibility

ID - rules

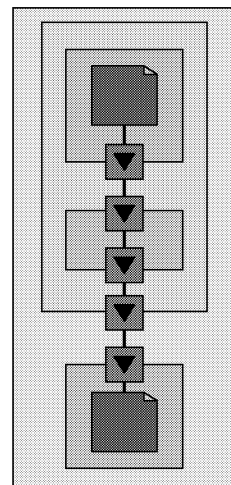


Interface Chains

- binding = connecting *modules* to *interfaces* to *interfaces* to *modules*.
- components only serve as *scope restrictors* during the binding.



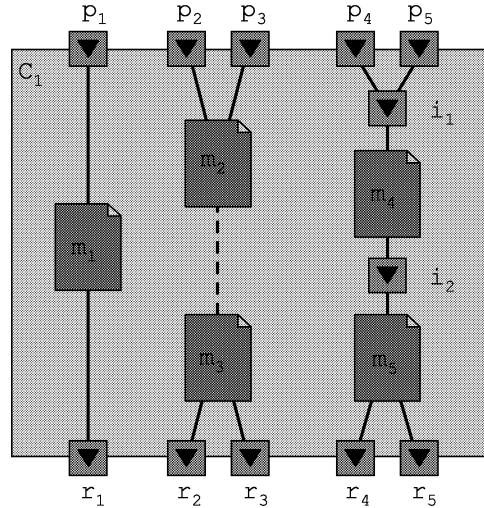
- each *tip* should be connected to precisely one base or module;
- each *base* may be connected to zero or more tips or modules;
- modules cannot be bound to modules (un-typed connection)!



Implementing Components

An example implementation of a component:

- m_1 implements p_1 in terms of r_1 ;
- m_2 and m_3 *communicate* outside Koala domain (in C using header file);
- m_4 and m_5 communicate through *private interfaces*.

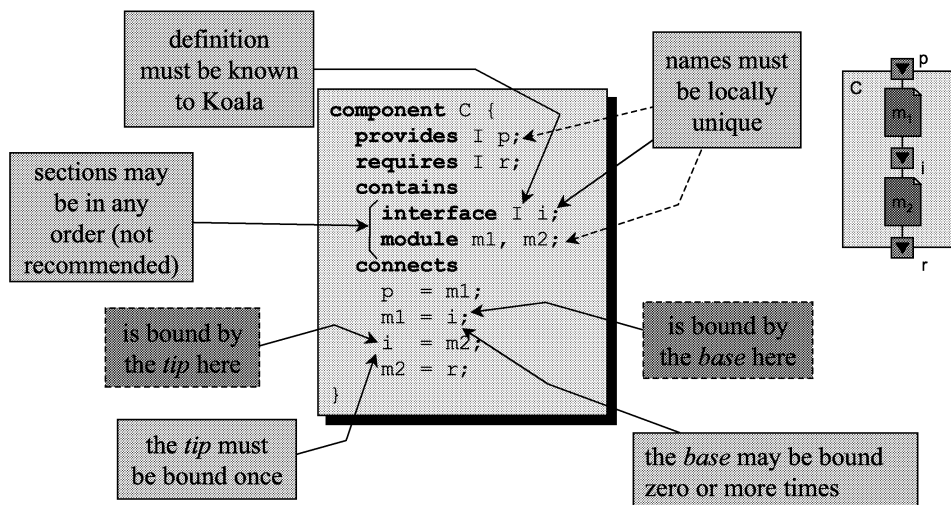


Koala, Sep 2003, RvO, 39

Koala Workshop

*

CD - private interfaces



Koala, Sep 2003, RvO, 40

Koala Workshop

Interface Compatibility

Once defined, an interface definition

may not be changed any more...

(unless all components referring to the definition are changed simultaneously)

but it is allowed to define *new* interfaces that are supersets (or subsets) of old interfaces.

Koala allows to connect such new interfaces to the old interfaces.

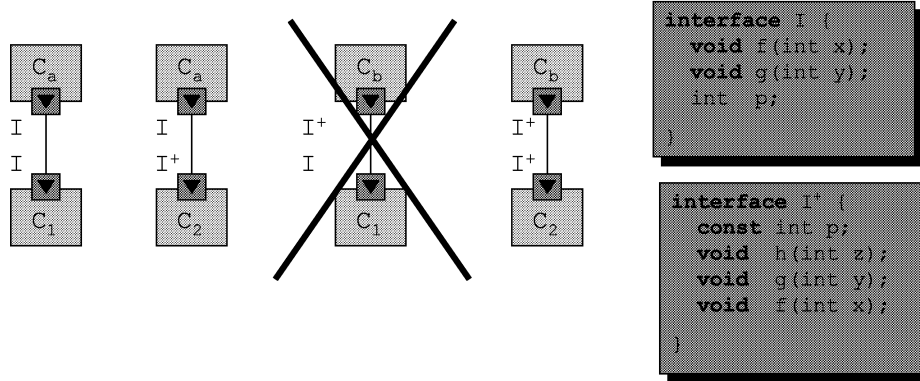
Compatibility Rules

Rule: a *tip* may be connected to *any* base, if for every function in the tip interface there is a function in the base interface with:

- the same function name
- the same result type
- the same parameter list, where each formal parameter has:
 - the same name
 - the same type
- the same or more “constantness”
- (for constants: have the exact same definition $2+3 \neq 3+2$)

Illustration

The pictures show how old and new interfaces may be connected.



Exercise: more on interfaces

```

interface I1 {
    int f(int x, int y);
    void g(char c);
}
    
```

```

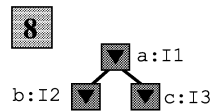
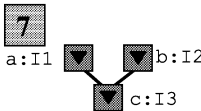
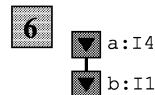
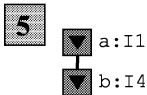
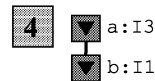
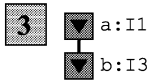
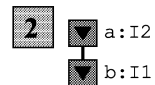
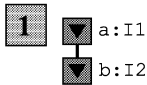
interface I2 {
    void g(char c);
    int f(int x, int y);
}
    
```

```

interface I3 {
    void g(char c);
    int h(void);
    int f(int x, int y);
}
    
```

```

interface I4 {
    int f(int x, int y);
    void g(char ch);
}
    
```



Your assignment
Which of 1..8 are ok?

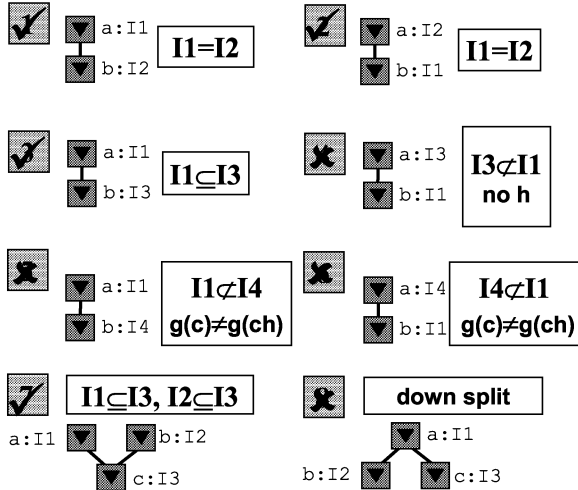
Answer: more on interfaces

```
interface I1 {
  int f(int x, int y);
  void g(char c);
}
```

```
interface I2 {
  void g(char c);
  int f(int x, int y);
}
```

```
interface I3 {
  void g(char c);
  int h(void);
  int f(int x, int y);
}
```

```
interface I4 {
  int f(int x, int y);
  void g(char ch);
}
```



Koala, Sep 2003, RvO, 45

Koala Workshop

4. Function Binding

1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

We discuss

- diversity interfaces
- native Koala expressions
- in-line expressions

Koala, Sep 2003, RvO, 46

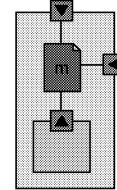
Koala Workshop

Implementing Functions

Rule: every function in every interface that is connected with the tip to a module should be implemented by that module.

Implementation can be:

- either in *C*,
- or in *Koala*



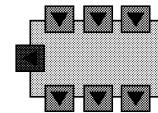
The latter is sometimes called *function binding*; its main purpose is to support diversity.

Diversity Interfaces

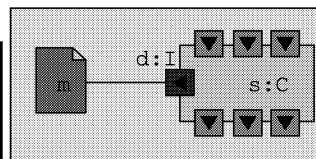
Component diversity may be controlled through *diversity interfaces*.

These are *requires interfaces*, allowing us to:

- assign values to diversity parameters using the binding mechanism;
- delay static/dynamic decision
- optimise on certain constant values.



```
interface I
{
  Bool Flag(void);
}
```



```
Bool s_d_Flag( void )
{
  return TRUE;
}
```

In C

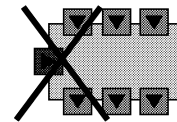
```
connects
  s.d = m;
within m {
  s.d.Flag() = true;
}
```

In Koala

Diversity Interfaces

Diversity interfaces are not provided interfaces
(using SetColor instead of required with Color)

- hard to optimize away
(now a #define suffices)
- provided needs notification
(propagate changes)
- initialisation problems
(when can a component use property)



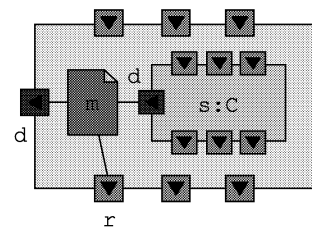
Diversity Spreadsheet

Compound components may have
diversity interfaces as well.

The *inner* diversity interfaces may be expressed in terms of constants, functions and parameters in the *outer* diversity interface.

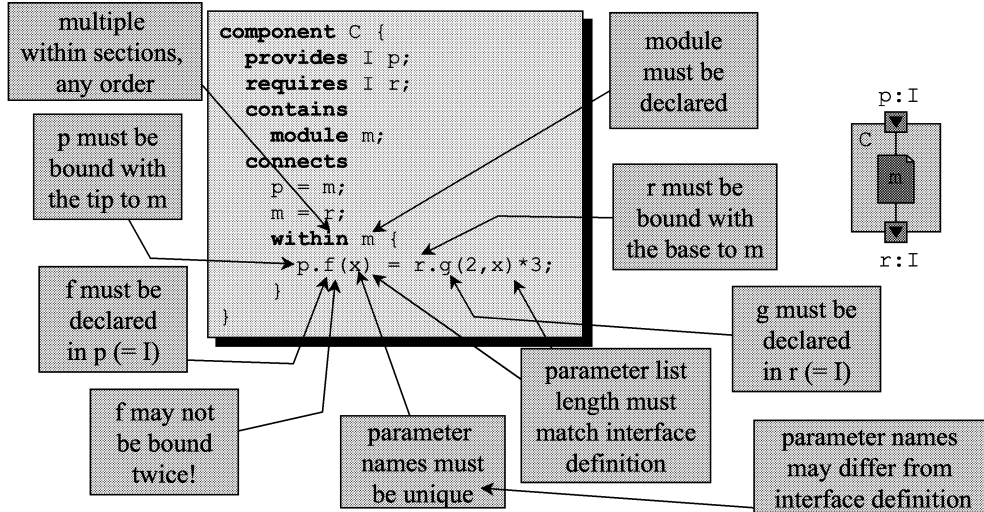
This allows to use different ‘languages’ for diversity depending on the level of decomposition.

Catch: re-evaluated every time

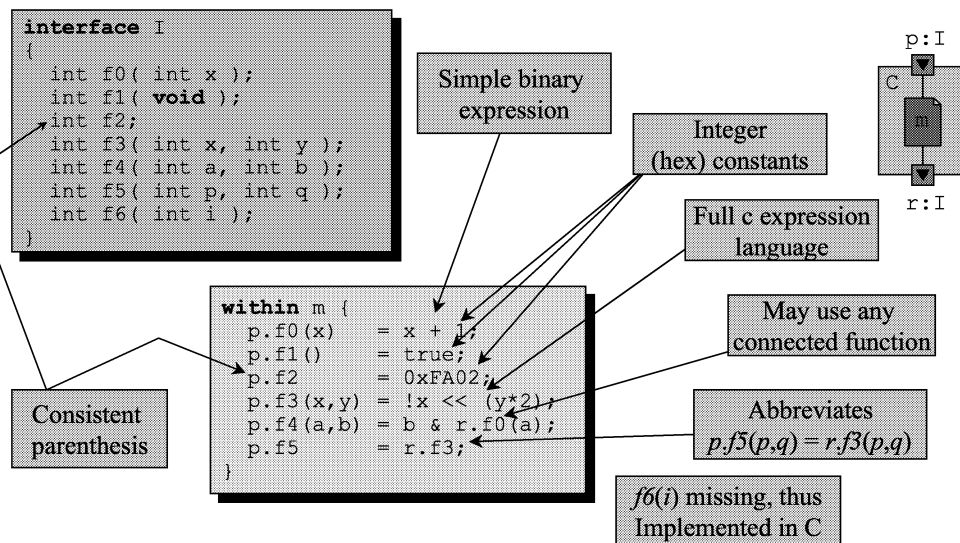


```
connects
s.d = m; m = d; m = r;
within m {
  s.d.f() = ! d.f();
  s.d.g() = - r.g();
}
```

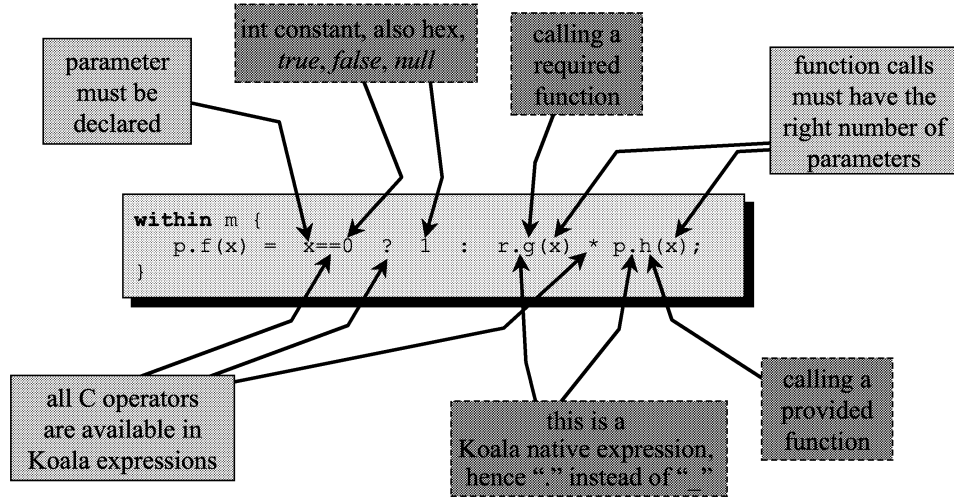

CD - within clause



CD – examples of within clause



CD – native expression

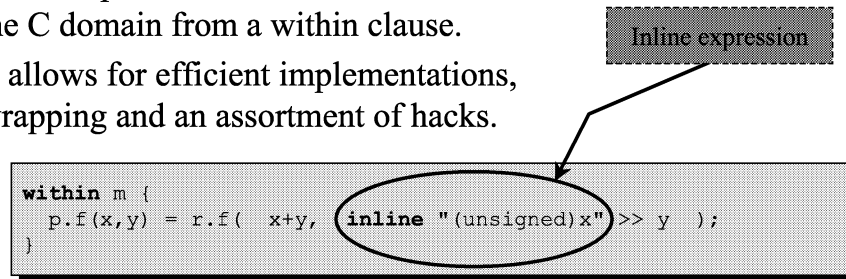


Inline

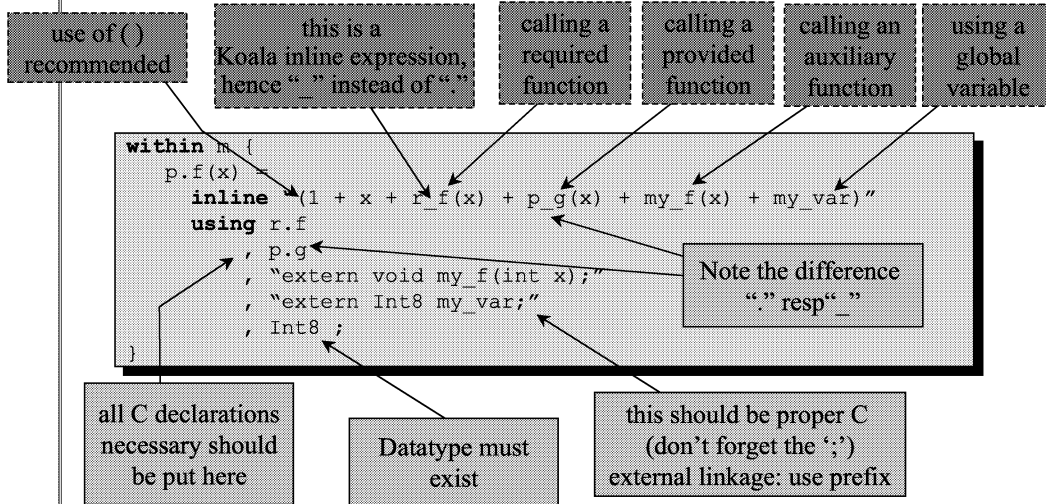
To achieve full expressional power in Koala, the *inline operator* was added.

The *inline operator* allows one to refer to the C domain from a within clause.

This allows for efficient implementations, wrapping and an assortment of hacks.



CD - inline expression



Inline expressions

Ingredients of inline expressions are

- constants (0, 1,... , NULL, TRUE);
- operators (+, &);
- foreign functions `legacy_func`;
- optimizations `my_var` and `my_func`.

Not needed, already possible in Koala

Make unique with component shortname (prefix)

Koala expression → macro

Koala expressions, native as well as inline ones, are mapped to macro's, not functions.

This makes them interesting from an efficiency point of view (slide+1, slide+2).

This makes them dangerous from a syntax point of view (slide+3).

This makes them problematic from a function-address point of view (slide+4).

Native expression - efficiency

```
interface IGetter // get
{
  int Val1( void );
  int Val2( void );
}
```

```
component CAaComponent
{
  specials
  prefix "aac";
  provides
  IGetter get;
  contains
  module m;
  connects
  get = m;
  within m {
    // get.Val1 in C
    get.Val2() = 3;
  }
}
```

```
component CBbComponent
{
  specials
  prefix "bbc";
  requires
  IGetter get;
  contains
  module m;
  connects
  m = get
}
```

```
/* Koala generated _ssc_m.h */
#define get_Val2 aac__get_Val2
extern int aac__get_Val2( void );
#define get_Val1 3
```

Macro!

Inline expression - efficiency

```
interface IAccess // acc
{
    int Get( void );
    void Set( int v );
}
```

```
#include "_ssc_m.h"
int ssc_MyVal;

/* provides IInit init */
void init_Init( void )
{
    ssc_MyVal = ... read nvm ...
}
```

```
component CSsComponent
{
    specials
    prefix "ssc";
    provides
    IInit init;
    IAccess acc;
    contains
    module m;
    connects
    init= m;
    acc = m;
    within m {
        acc.Get() =
            inline "ssc_MyVal"
            using "int ssc_MyVal;";
        acc.Set(v) =
            inline "(ssc_MyVal=v)"
            using "int ssc_MyVal;";
    }
}
```

Inline expression - syntax

When stubbing, don't write `p.f(v) = inline " "`

- a function is an expression,
- and the empty expansion is not
- example `x= (r_f(3), 5)`
- won't compile

```
within m {
    // Use either old style:
    p.f(v) = inline "(void)0";

    // or better, the new style:
    p.f(v) = void;
}
```

Don't write `p.Dup(v) = inline "v+v"`

- inline is implemented as macro
- so you get the macro-()-hell
- example `x= 3*r_Dup(5)`
- x is now 20, not 30

```
within m {
    p.Dup(v) = inline "(v+v)";
    // Koala parenthesizes args!
}
```

Expression - addressing

Sometimes it is desired to take the address of a required function.

- this not allowed, because
- it could be a macro!
- if needed, use addressable
- however: ROM size increase

```
static Function mine;
void SomeFunc( Bool b )
{
  if( b )
    mine= &r1_f;
  else
    mine= &r2_f;
}
```

```
requires
  IInterface r1;
  IInterface r2;
contains
  module m;
connects
  m = r1;
  m = r2;
  within m {
    addressable r1.f, r2.f;
  }
```

Implementation choices

For the implementation of a function we have three possibilities:

- C (a function in a c file)
- Koala (an expression in a within in a cd file):
 - native
 - inline

```
interface I
{
  int f( int x );
}
```

```
#include "c_m.h"
int p_f( int x )
{
  return x+x;
}
```

```
within m {
  p.f(x) = x+x;
}
```

```
within m {
  p.f(x) = x + inline "(char)x";
}
```

5. Constants

1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

We discuss

- Constant folding
- Using constantness
- **CONSTANT** macro
- **const** operator and declarator

Constant Folding

Koala understands native expressions, their constants, operators and binding.

Koala can optimize these expressions, this is referred to as constant folding.

Koala can not fold c-implementations or inline expressions.

Constant Folding Examples

- $i.f(x) = 3 + 4$ has value 7
- $i.g(x) = 5 + i.f(x)$ has value 12

- $i.h(x) = x + 1$ is not constant folded
- $i.p(x) = i.h(5)$ has value 6

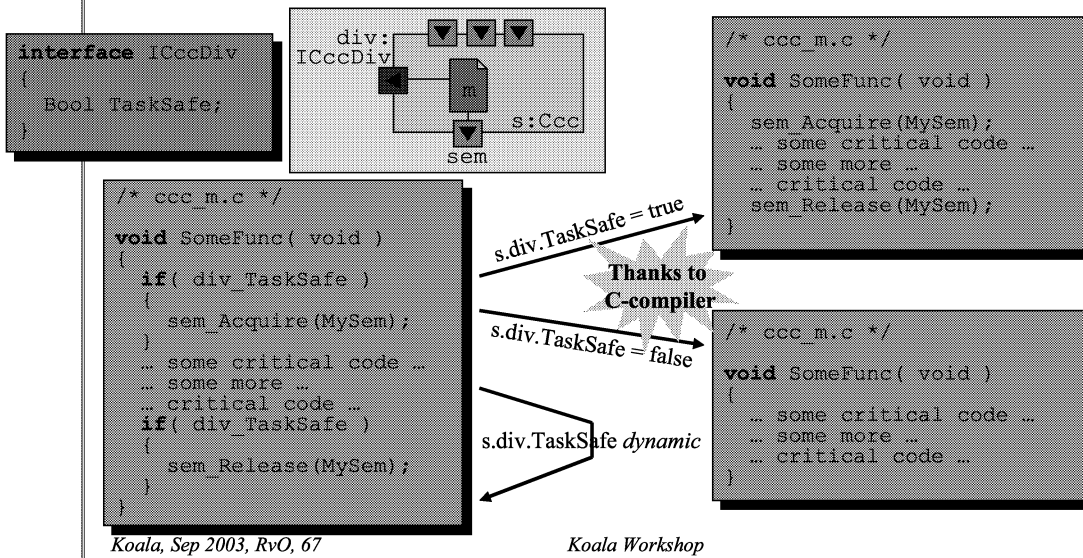
- $i.b = 0 \ \&\& \ true$ has the value *false*
- $i.c() = i.b ? j.d() : j.e()$ optimizes $i.c()$ to $j.e()$

Using constantness

When a function is constant certain optimizations are possible:

- Removing dead statements
- Removing dead definitions
- Using static arrays, switch/case
- Removing dead components

Some optimizations go automatically, others need help from you.

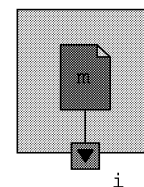


Whenever Koala determines that a function f in interface i is boolean or integer constant, it generates a macro with the name $i_f_CONSTANT$ that has the value of the function as expansion.

```

#ifndef i_f_CONSTANT
#define i_f_CONSTANT
/* it's true! */
#else
/* it's false! */
#endif
#else
/* maybe true, maybe false */
#endif

```



```

#ifdef div_TaskSafe_CONSTANT
#if div_TaskSafe_CONSTANT
    Semaphore MySem;
#else
    /* no semaphore needed */
#endif
#else
    Semaphore MySem;
#endif

void init_Init( void )
{
    if( div_TaskSafe
    {
        MySem=sem_SemCreate();
    }
}

```

Here Koala generates

#define div_TaskSafe_CONSTANT 1

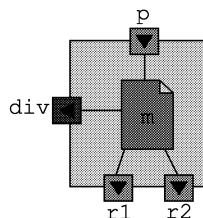
Here Koala generates

#define div_TaskSafe ((Bool)1)

Use i_f_CONSTANT for pre-processor,
and i_f for C-compiler

In a Koala expression, we can use the const operator.

It evaluates to true when Koala can compile-time fold the operand to an integer constant.



```

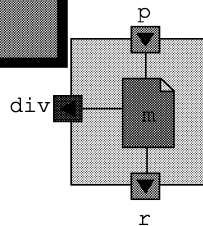
component CComponent
{
    provides
        I p;
    requires
        I r1, r2;
        IDiv div;
    contains
        module m;
    connects
        p = m;
        m = r1; m = r2; m = div;
    within m {
        p.f(x) = (const div.A) ? r1.f(x) : r2.f(-x);
    }
}

```

The const *declarator*

Sometimes, a required function (typically an interface parameter) should be compile-time constant, for the C-code to be correct. Use the const declarator.

```
interface IDiv
{
  int Size;
}
```



```
component CComp
{
  provides
  I p;
  requires
  I r;
  IDiv div;
  contains
  module m;
  connects
  p = m;
  m = r;
  within m {
    const div.Size;
  }
}
```

```
#include "_CComp_m.h"
int MyTable[ div_Size ];
void init_Init( void )
{
  int i;
  for(i=0; i<div_Size; i++)
    MyTable[i]= i;
}
```

Issue!

Dead components

When a component is dead this is detected by Koala's reachability algorithm (see last chapter).

Switches (see last chapter) help making components dead.

6. Optional Interfaces

1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

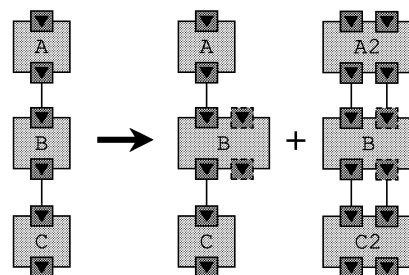
We discuss

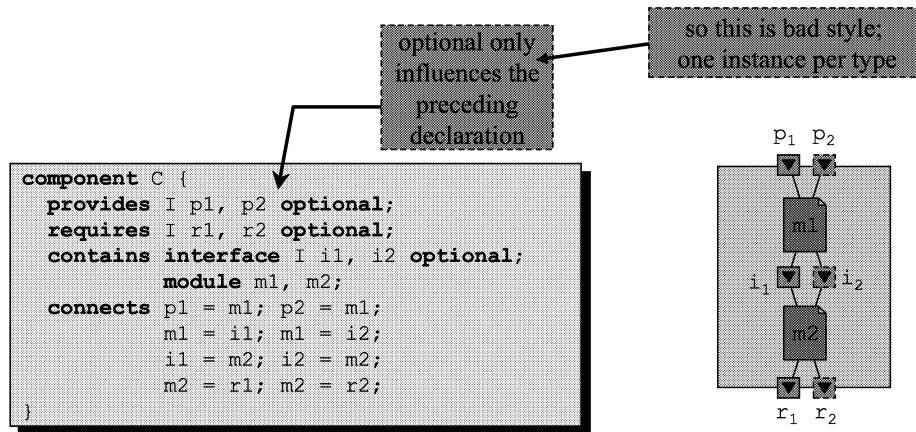
- optional interfaces
- iPresent

Evolving Components

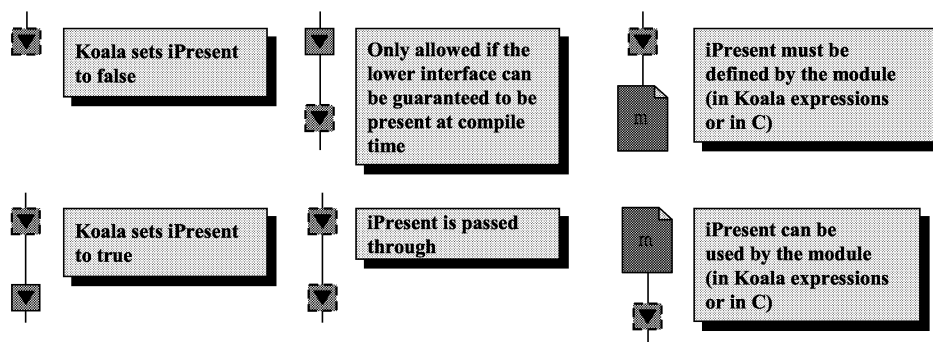
Interfaces *must* be connected at the tip, unless they are declared *optional*.

Optional interfaces allow components to evolve over time, without having to change existing configurations.



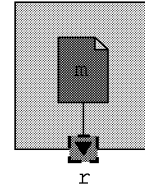


Every optional interface is augmented with a nullary integer function `iPresent` (with a `_CONSTANT` macro).



iPresent optimised away

- If *iPresent* is statically known the compiler will optimise
 - remove guard
 - remove either then or else part
- If *iPresent* is dynamic code still works



```
if( r_iPresent() )
{
    /* can make call to r */
    x= r_SomeFunc();
}
else
{
    /* do it yourself */
    x= 0;
}
```

Exercise

Hello you
(twice)

7. Switches

1. Concepts
2. The Basic Model
3. More on Interfaces
4. Function Binding
5. Constants
6. Optional Interfaces
7. Switches

We discuss

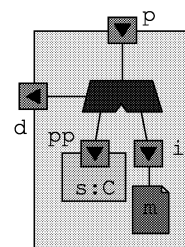
- switches
- compatibility and optionality
- reachability
- ICONNECTED

Flexible Connections

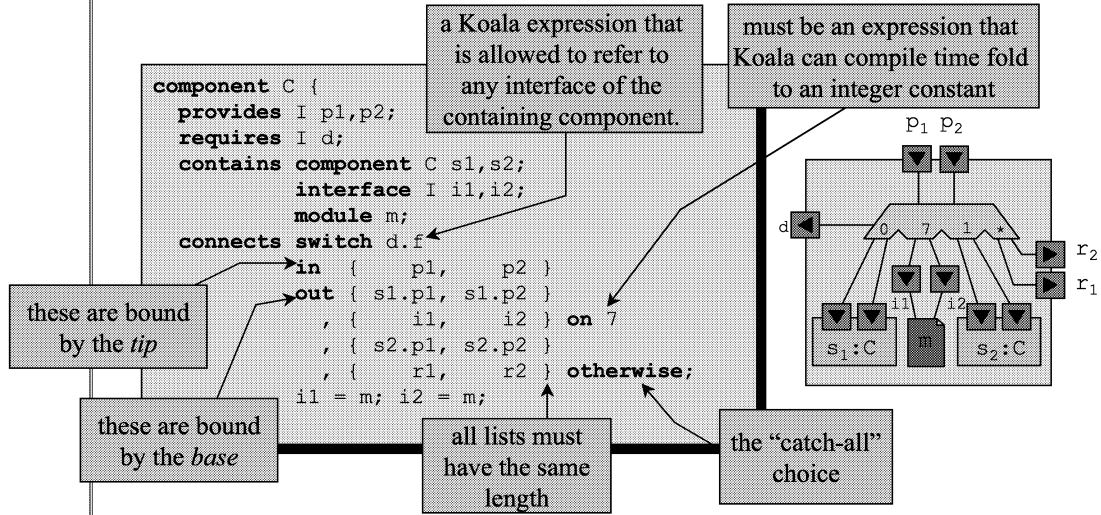
A *switch* allows to create bindings that depend on the value of certain functions.

In the example, *p* is connected through the switch to either *pp* or *i*, depending on the value of the control function in interface *d*.

Koala optimises switches if the setting is known at compile time, and generates switch code otherwise.



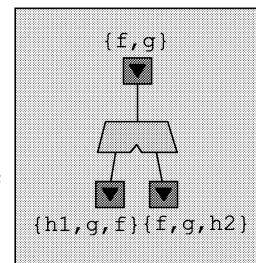
Switch - Well Formedness



Interface Compatibility

A switch is a special form of *binding*.

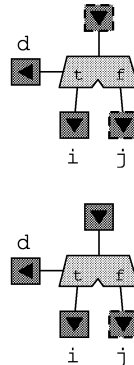
The compatibility rules are the same. This means that *any* function in *any* interface in the **in** clause must have an implementation in the *corresponding* interface in *every* **out** clause!



Optional Interfaces

Rules for connecting optional interfaces can be derived from the general binding rules:

```
switch d.f
  in { p }
  out { i } otherwise
    , { j } on false
;
```



Koala calculates iPresent to be:
 $d.f() ? true : j.iPresent()$

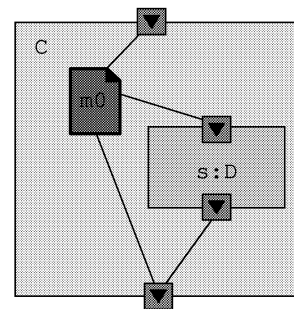
Is allowed, provided that Koala can guarantee at compile time that:
 - $d.f()$ holds, or
 - $j.iPresent()$ holds

Reachability

Some functions are called by 'Magic', i.e. outside the Koala domain:

- `main()`
- interrupt handlers

If a module contains such a function, declare it as present



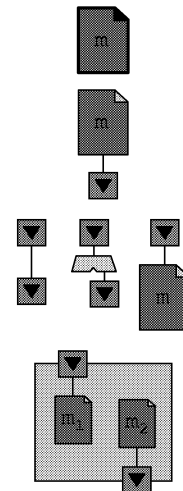
```
component C {
  ...
  contains
    module m0 present;
  ...
}
```

Reachability

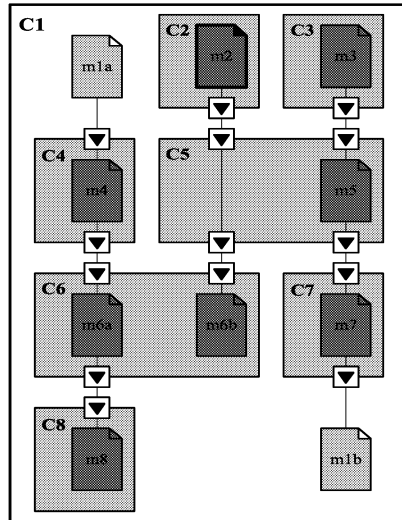
- Koala generates header files for all modules in reachable components.
- Informally: a component is reachable if it is (indirectly) called from a present module.

Reachability

- modules marked **present** are reachable;
- if a module is reachable, then an interface bound with its base to the module is reachable;
- if an interface is reachable, then an interface or module bound to its tip (directly or via a switch) is reachable;
- a component is reachable if at least one of its modules is reachable;
- a module is reachable if its container component is reachable.



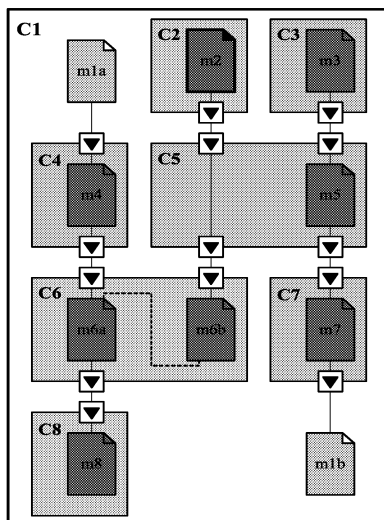
Reachability



We build C1.
Which components are reachable from C1..C8?
Which modules are built and linked from m1a..m8?

Reachability

Module m2 is present.



Koala assumes a call from m6b to m6a

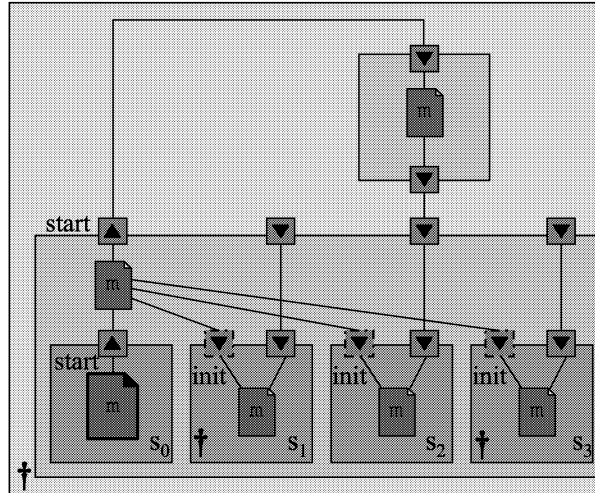
Note that we build component C1, but that C1 is not reachable!

Reachability

- However, reachability stops when reaching an optional interface
- This allows us to have optional initialisation

```
void s0_start_Init()
{
    if( s1_init_iPresent() )
        s1_init_Init();
    if( s2_init_iPresent() )
        s2_init_Init();
    if( s3_init_iPresent() )
        s3_init_Init();
    start_Init();
}

```



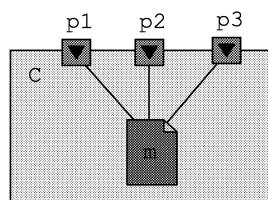
Koala, Sep 2003, RvO, 89

Koala Workshop

ICONNECTED

For every interface *i* bound with the tip to a module, a macro *i_ICONNECTED* is generated *if* that interface is bound with the base (possibly through other interfaces) to a reachable module.

This macro can only be used in C; there is no native Koala equivalent.



```
/* m.c */
#ifdef p1_ICONNECTED
/* functions for p1 */
int p1_f(int x) {
    return x+x;
}
#endif

```

Koala, Sep 2003, RvO, 90

Koala Workshop

Henrik Thane

Mälardalen University

Monitoring Testing and Debugging of Embedded Real-Time Software

Dr. Henrik Thane
Mälardalen University

MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

© Henrik Thane 2003 henrik.thane@mdh.se

Who am I?



Henrik Thane is an Assistant Professor at the Department of Computer Engineering, Mälardalen Real-Time Research Centre. Henrik has both an industrial and academic background. He received a Ph.D. from the Royal Institute of Technology in Stockholm and has worked as a programmer and consultant in the real-time and embedded systems area for several years. In addition to research he has during the last seven years worked as an expert consultant for the industry and given numerous industrial courses on design and verification of software in safety-critical computer based systems. Henrik's research interests are design and verification of safety-critical systems, monitoring, debugging and testing of (distributed) real-time systems, as well as design of real-time operating systems, and scheduling. The leading star is that the results should be of practical use.
Henrik.thane@mdh.se

Henrik Thane is also the CEO and President of ZealCore Embedded Solutions AB, a company focused on bringing state-of-the-art research to the industry. Among the products provided are the unique TimeMACHINE and BlackBox debugger for embedded systems, but also design and analysis tools for distributed automotive applications. ZealCore provides professional services in terms of consulting and education in the safety-critical real-time systems area.
Henrik.thane@zealcore.com



www.zealcore.com



www.mrtc.mdh.se

© Henrik Thane 2003 henrik.thane@mdh.se

Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

The Problem

Releasing Reliable Software on Schedule is Almost Impossible ...

Limited development and
QA resources

Increasing
complexity
(more features)

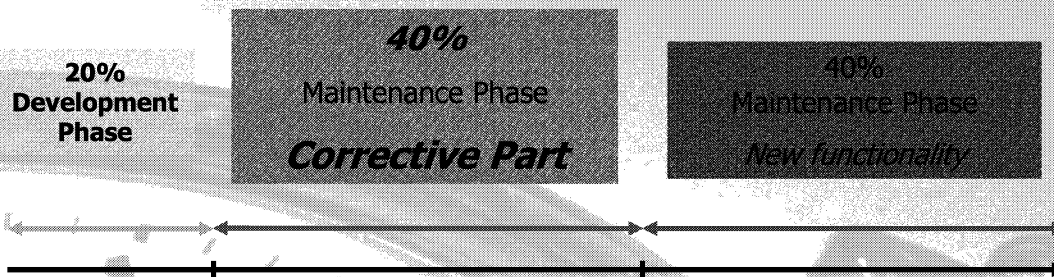
Shortening
product
lifecycles

... and Management Has Almost No Visibility Into Software Quality Before Release

Testing Software is a Bottleneck

- Most testing efforts evaluate only 30% to 40% of a software system. [Standish Group]
- “Bug fixing is at least 30% of development effort and increases with software complexity” [Capers Jones]

Distribution of Life Cycle Cost

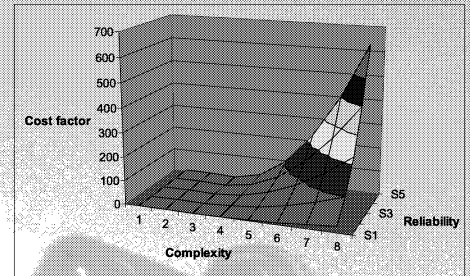
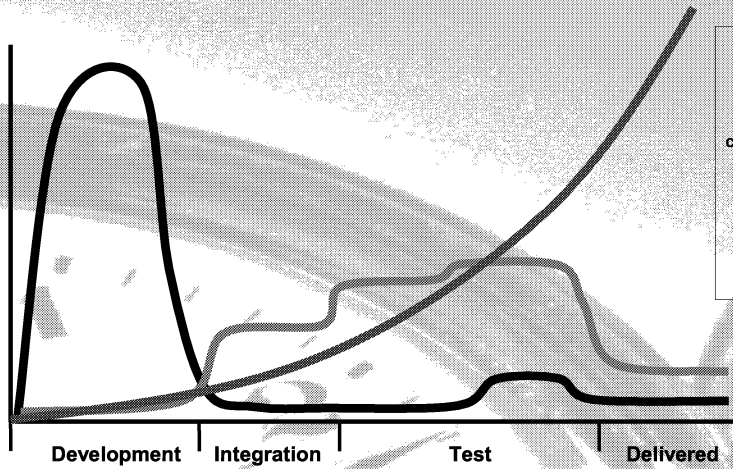


- ▶ Corrective costs > 2 x Development costs [NASA, Bohem]
- ▶ US bug cost > \$59 Billion/Year (Excluding mission critical systems) [NIST 2001]

The Cost of Defects

Finding Bugs Late is very Expensive and Time-Consuming

- No Defects Created
- No Defects Found
- Cost to Repair



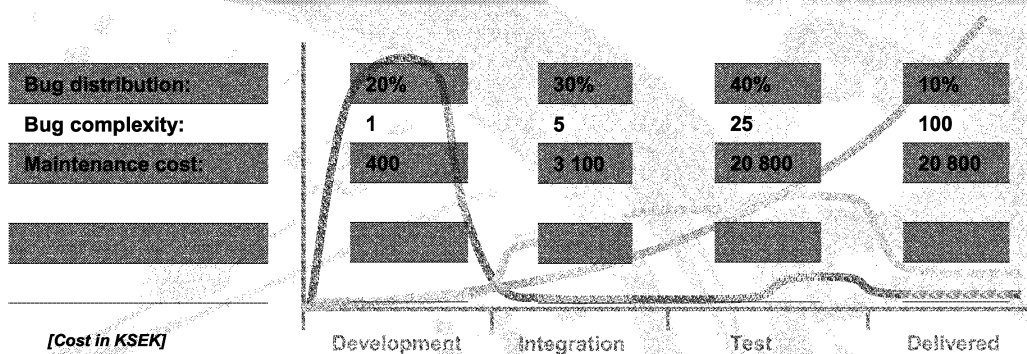
[Capers Jones]

► 80% of software projects are released late and over budget [PriceWaterhouseCoopers]

The Cost of Defects

Facts:

- Software team: 100 people
- Maintenance team (40%): 40 people
- Cost: 600 SEK/h
- Maintenance cost / year: 45 MSEK

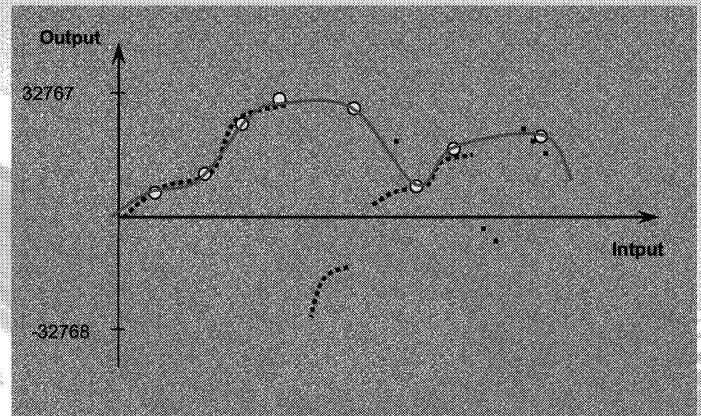


[Cost in KSEK]

Fundamental problem: Discontinuity

Fundamentally hard to test software in general.

- Cannot interpolate or extrapolate.
- "Can only show the presence of errors not their absence" - Dijkstra
- 40 sequential if-then-else statements $\rightarrow 2^{40}$ paths takes 34 years 1 test/ms.



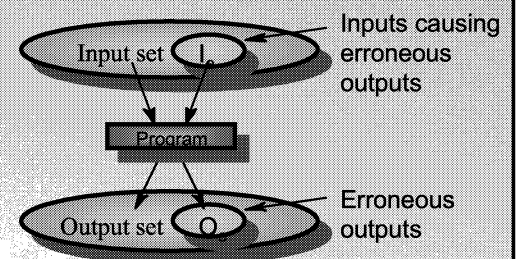
Testing programs, debugging programs ?

Fault \rightarrow Error \rightarrow Failure

- When we *test* a program we look for *failures*
 - A behavior violating the specification is a failure
- When we *debug* a program we look for the *faults* causing the failures.

Systematic failures, occur if and only if:

1. Execute fault
2. The execution of the fault leads to an infection (corrupted data). The error.
3. Infection propagates to output.



Reproducibility is fundamental

- Regression testing
- Cyclic debugging



Testing in general

- Establishes *presence*, not *absence* of defects!
- Tester's goal is to make program fail!
- Ideally performed independently
- Normally cannot test all possibilities

Reliability improvement

Basic testing techniques

Black box or White box testing
specification based or implementation based

Functional testing (Black-box)

Coverage testing (all possible inputs)

Random testing

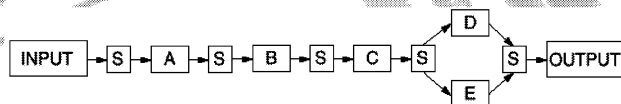
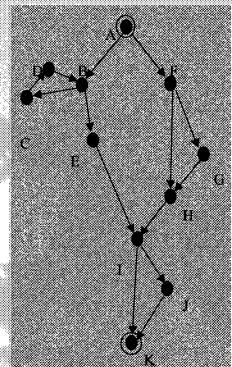
Structural tests (White-box).

Equivalence class testing

Boundary value testing

Transaction testing

Fault injection



Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

Testing components and architectures

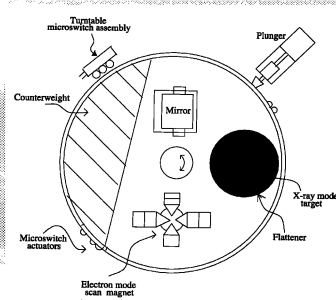
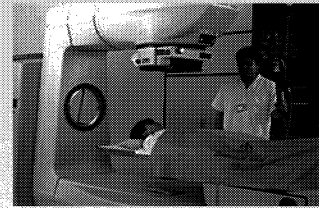
Why Component Reuse?

- The arguments for reuse of software (components) are usually
 - Improved productivity (reusing code),
 - Outsourcing (let someone who's better do it),
 - Higher reliability (inherit "quality").
 - Assume that verification of the components can be eliminated or reduced.
 - Expensive and catastrophic experiences have however shown that it is not so simple, e.g., Ariane 5, and Therac 25.

Reuse: A growing problem?

Therac-25

- Computer controlled radiation therapy machine.
- 6 accidents 1985 – 1987. Three people died of radiation burns.
- Error in the Man-Machine-Interface + 6bit counter turned 0 + hardware interlock removed



Reuse: A growing problem?

The controlling software was written in ADA but is in essence equivalent with the following C code:

```
short s;  
double d;  
...  
s = d;
```

This happened:

A 64 bit float was truncated to a 16 bit integer in a "non-critical" software component.

When the integer overflowed the ADA program ran an exception handler that shut down the entire system, including the control software of the rocket.

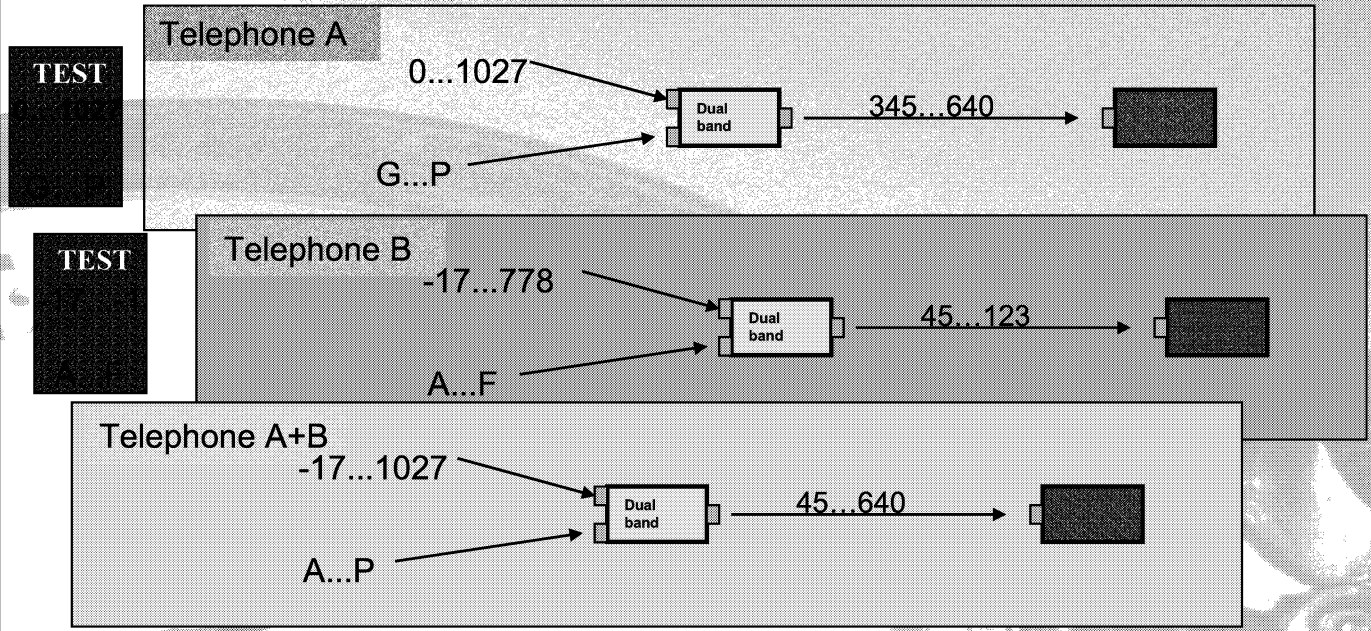
The system had two redundant computers - both running the same software.

The erroneous component (a method) was inherited/reused from Ariane 4 and had no practical use in Ariane 5. As it had no function in Ariane 5 its exception handler was removed. (BIG MISTAKE)

Lesson learned???

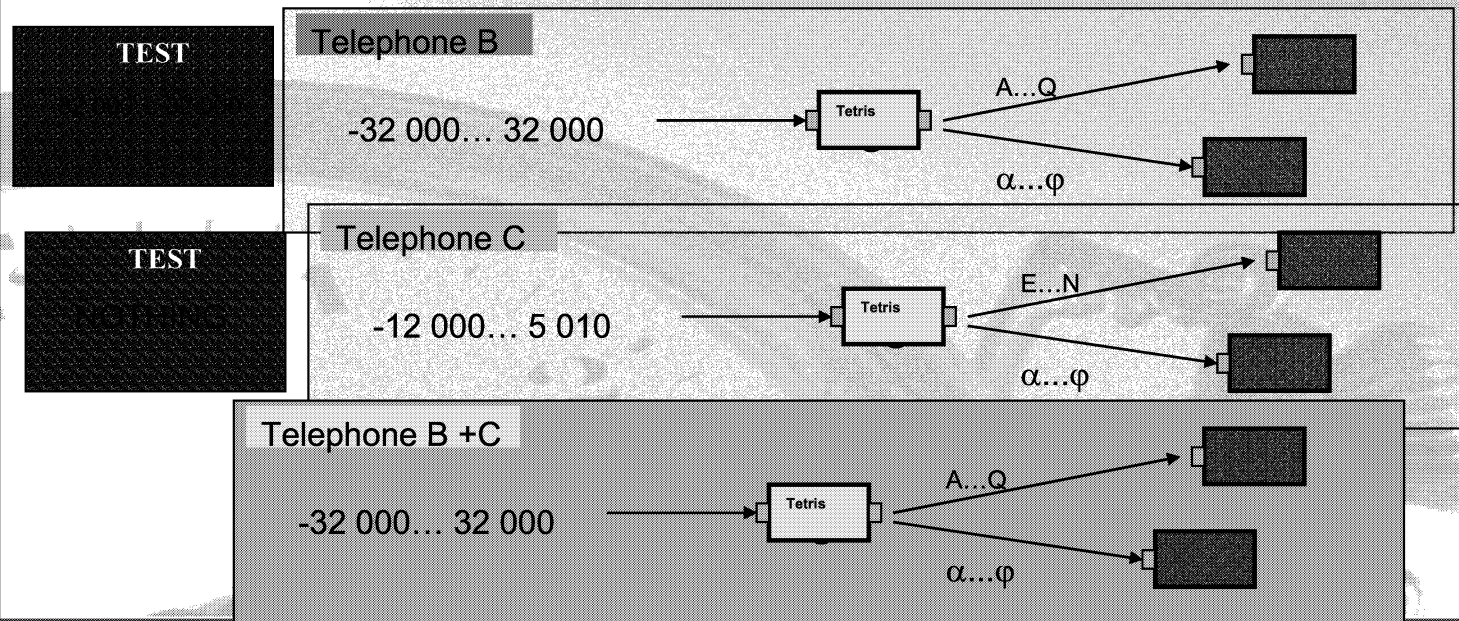
Controlled Reuse and Controlled Testing

Non overlapping input-output domains



Controlled Reuse and Controlled Testing

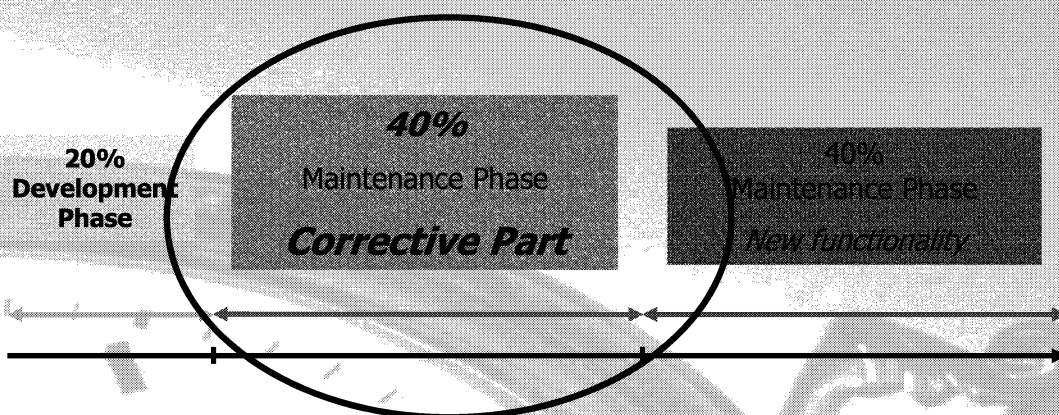
Overlapping input-output domains



Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

Distribution of Life Cycle Cost (Again)



- ▶ Corrective costs > 2 x Development costs [NASA, Bohem]
- ▶ US bug cost > \$59 Billion/Year (Excluding mission critical systems) [NIST 2001]

The constructive approach: Design and measure for testability !!

Testability = The inverse to test effort

Testability factors

- **Coverage**
 - **Size of state-space to explore**
 - **Defined by exploration method (test method)**
 - Observability
 - Data extraction
 - Intrusiveness
 - Controllability
 - Determinism
 - Reproducibility
- The number of possible inputs to the system
 - E.g., INT32 f(INT32 a, INT32 b, INT32 c) --> 2^{96}
 - The number of paths through a program

A Metric for Testability: The DRR

The Domain Range Ratio (DRR):

- The DRR of a unit (module, function, operation) is the ratio between the cardinality of the domain (input) to the cardinality of the range (output).
- A high value of DRR indicates a high potential of the module to hide errors ==> *low testability*.

```
function F (in integer X, Y; out boolean B);  
begin  
-----  
if C then A:=( X+ Y)* K1 else A:=( X+ Y)* K2;  
B:= A> 1000;  
end;
```

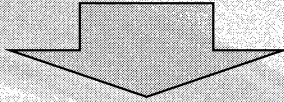
DRR (F) = ∞ : 2

A Metric for Testability: The DRR (Cont'd)

A high DRR indicates modules of the program that are less likely to expose existing errors

- *Errors are hidden because of "low observability"*

These modules have to be tested thoroughly



OR

Testability has to be improved by

1. Adding new output parameters ==> increase observability
2. Introducing ASSERTIONS ==> Defensive programming/ BIST

A Metric for Testability: The DRR (Cont'd)

Improved testability

```
function F( in integer X, Y; out boolean B; out integer A);  
begin  
-----  
    if C then A:=( X+ Y)* K1 else A:=( X+ Y)* K2;  
    B:= A> 1000;  
end;
```

```
function F (in integer X, Y; out boolean B);  
begin  
-----  
    if C then A:=( X+ Y)* K1 else A:=( X+ Y)* K2;  
    ASSERT( cond( A), `erroneous state in function F");  
    B:= A> 1000;  
end;
```


The constructive approach: Design and measure for testability !!

Testability = The inverse to test effort

Testability factors

- **Coverage**
 - **Size of state-space to explore**
 - **Defined by exploration method (test method)**
- Observability
 - Data extraction
 - Intrusiveness
- Controllability
 - Determinism
 - Reproducibility

```
int f(int a)
{int x;
  if (...) {
    ...;
    if (...){
      x = a-1; /* x=a+1; Error*/
      x = x div 30000;
      return x;
    }
    ...;
  }
  ...;
}
```



Probability of failure is: $0,25 * 1 * 2 / 30000 = 0,0000166$.


The constructive approach: Design and measure for testability !!

Testability = The inverse to test effort

Testability factors

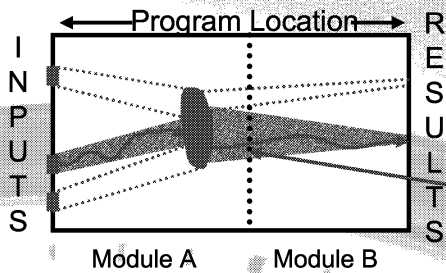
- **Coverage**
 - **Size of state-space to explore**
 - **Defined by exploration method (test method)**
- Observability
 - Data extraction
 - Intrusiveness
- Controllability
 - Determinism
 - Reproducibility

```
float f(float a)
{float x;
  if (...) {
    ...;
    if (...){
      x = a-1; /* x=a+1; Error*/
      x = x / 30000;
      return x;
    }
    ...;
  }
  ...;
}
```



Probability of failure is: $0,25 * 1 * 1 = 0.25$

Another picture: Error Size and Location



```
int f(int a)
{int x;
  if (...) {
    ...;
    if (...) {
      x = a-1; /* x=a+1; Error*/
      x = x div 30000;
      return x;
    }
    ...;
  }
}
```

To find an error

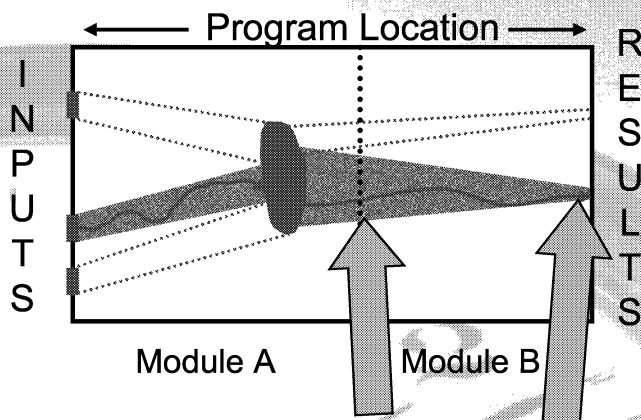
- Choose a test case which executes the erroneous statements
- Notice the failure caused by the error

Large errors easier to find

Size varies with location

- The error is larger at the boundary of module A than B

Design for High Testability (Increased observability) Defensive programming

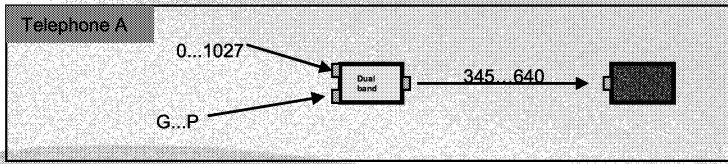


```
assert <pre-condition>;
  statement 1;
  .
  .
  .
  statement n;
assert <post-condition>;
```

Action
Exception handling
Notify operator

Easier to detect error here, than here

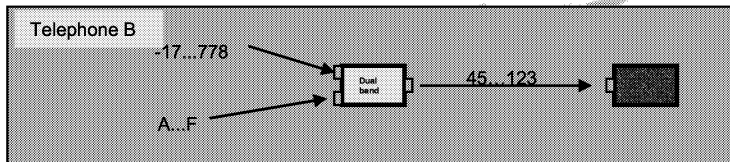
Design for High Testability (Increased observability) Defensive programming (Cont'd)



```

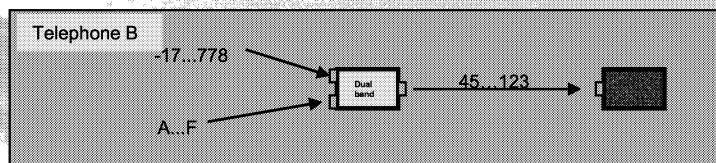
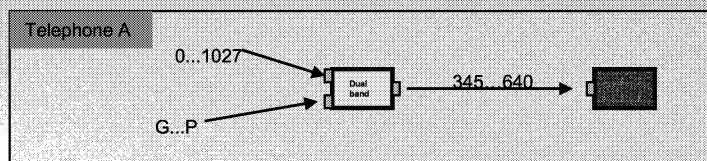
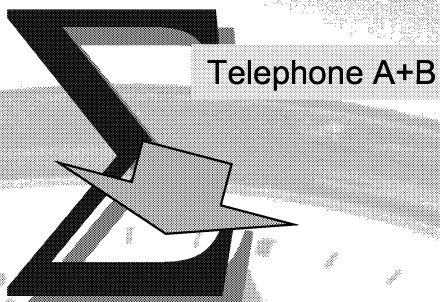
assert ( (0 ≤ input1 ≤ 1027) && ("G" ≤ input2 ≤ "P") ) // pre-condition
statement 1;
.
.
.
statement n;
assert (345 ≤ output ≤ 640) // post-condition
    
```

New instantiation



What happens???

Design for High Testability (Increased observability) Defensive programming (Cont'd)



```

assert ( (-17 ≤ input1 ≤ 1027) && ("A" ≤ input2 ≤ "P") ) // pre-condition
statement 1;
.
.
.
statement n;
assert (45 ≤ output ≤ 640) // post-condition
    
```

Design for High Testability

BIST - Built In Self Test

Include test cases in Object structure

```
Class class-name
{
    // Interface
    Data declaration;
    Constructor declaration;
    Destructor declaration;
    Function declarations;
    Test declarations;
    // Implementation
    Constructor;
    Destructor;
    Functions;
    TestCases;
}BISTObject;
```

Use e.g.,

```
BISTObject::TestCase1
BISTObject::TestCase2
...
BISTObject::TestCaseN
```

- Allows inheritance of test cases
- Eliminates the need for reanalysis and redesign of the test cases.
- Allow you to validate the reused implementation in a "new" environment.

Code overhead??

Key Points

- **Testing software is fundamentally hard**
- **Design for high testability**
 - Increase the observability
 - Assertions, BIST
 - Domain to range ratio (increase the output)
 - There exist a fundamental tradeoff between error tolerance and testability
- Reuse of software does not necessarily mean "no" verification rather the opposite
- Test of architectures means identifying all possible uses and different instantiations of the components belonging to the architecture
 - Incrementally build assertions and BIST
- **You always test against the specification**
 - 50% of system failures belong to erroneous specifications

You always test against the specification



- Systematic faults
- All wrong despite V&V

'The radar system of HMS Sheffield identified an incoming Exocet missile as non-Soviet and thus friendly. No alarm sounded. The Ship sunk with substantial losses of life.' [SEN1]

Key points 2

To "test in" reliability and confidence in software is for failure rates less than 10^{-4} f/h an impossible task in practice

- Software is discontinuous
- You do not find specification errors

Safety

- Accidents seldom happen in a manner anticipated by the designer
- Accidents seldom happen due to component failure of types easily predicted by reliability modeling and testing

Does not mean: DO NOT TEST, only that there is a limit to the confidence achievable by testing

There are also limits to "static" analysis - need test for validation of:

- Models
- Assumptions, forming the basis/foundation for mathematical techniques.
- Performance, timing, overload, etc.

Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring, Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

Monitoring, Testing and Debugging of RTS

- Testing and Debugging of Embedded Real Time Software is A "black art"
 - Ad hoc methods and techniques.
 - Ineffective and inadequate.
- Huge costs associated with validation of embedded applications.
 - Despite this, most difficult errors are discovered extremely late in the testing process, making them even more costly to repair.

Monitoring, Testing and Debugging of RTS

Testing

Executing a piece of software in order to reveal errors

- A substantial portion of the validation process.
- Development of test procedures, generation and execution of test cases.

Debugging

This is concerned with locating and correcting the cause of an error once it has been revealed.

- Developer must recreate exact execution scenario.
- Same instruction sequences
- All environmental variants must be accounted for.

Monitoring, Testing and Debugging of RTS

- Correct execution of embedded applications absolutely critical.
- **Testing and Debugging**
Greatly restricted by embedded systems, with constraints such as:
 - Concurrent Designs
 - Real-time constraints
 - Embedded target environments
 - Distributed hardware architectures
 - Device control dependencies
- These restrict execution visibility and control.
- **Target environment**
 - grossly inadequate computing resources.

Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring, Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

Monitoring

“In software engineering, monitoring is the recording of specified event occurrences during program execution in order to gain runtime information that cannot be obtained merely by studying the program text.”

Multitasking and distributed systems
The probe effect

The Lost Update Problem

Transaction T:

Bank\$Withdraw(A,4)
 Bank\$Deposit(B,4)

 Balance = A.Read() \$100
 A.Write (balance -4) \$96

Balance = B.Read() \$200

B.Write (balance +4) \$204

Transaction U:

Bank\$Withdraw(C, 3)
 Bank\$Deposit(B, 3)

 Balance = C.Read() \$300
 C.Write (balance - 3) \$297

Balance = B.Read() \$200

B.Write (balance +3) \$203

Multitasking and distributed systems
The probe effect

The Lost Update Problem

Transaction T:

Bank\$Withdraw(A,4)
 Bank\$Deposit(B,4)

 Balance = A.Read() \$100
 A.Write (balance -4) \$96

Probe →

Log (A.Read(), B.Read(), balance)

Balance = B.Read() \$203

B.Write (balance +4) \$207

Transaction U:

Bank\$Withdraw(C, 3)
 Bank\$Deposit(B, 3)

 Balance = C.Read() \$300
 C.Write (balance - 3) \$297

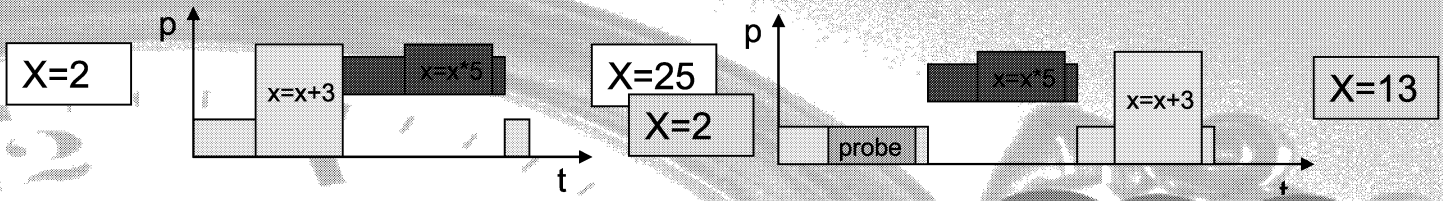
Balance = B.Read() \$200

B.Write (balance +3) \$203

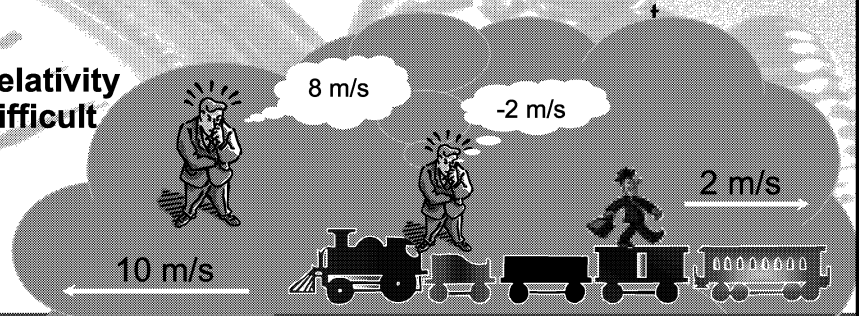
Monitoring

Embedded real-time systems are hard to observe

- Few interfaces to the outside world
- Observations may change the dynamic behavior (the probe effect)

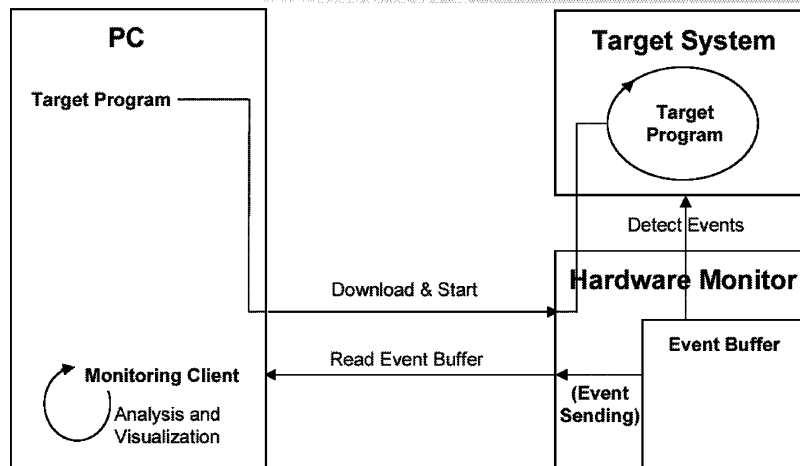
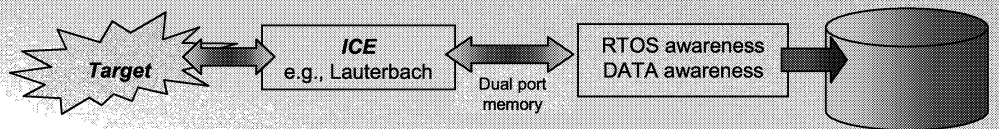


- In distributed real-time systems relativity makes consistent observations difficult
 - No common clock
 - Dependent on clock synchronization

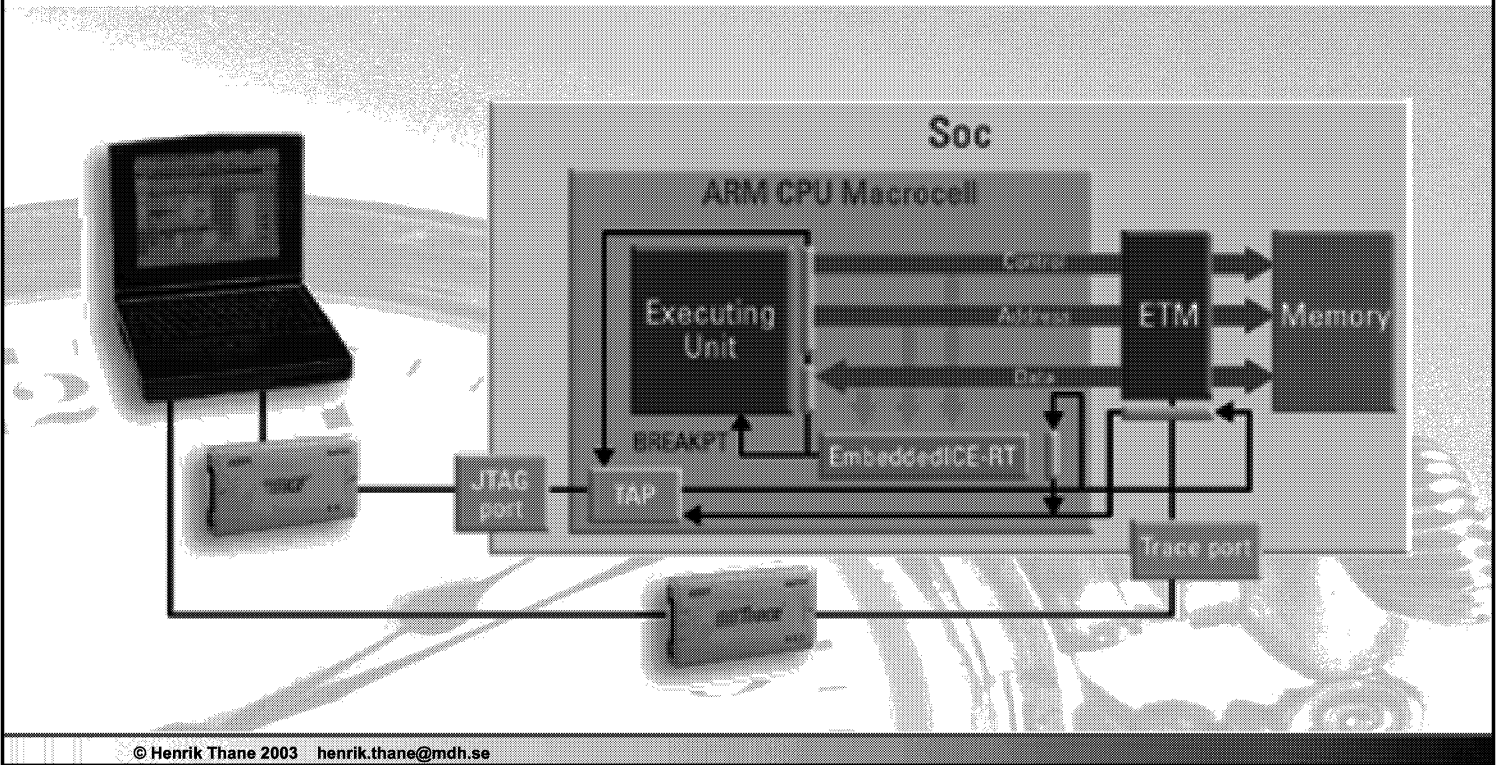


Non Intrusive Hardware Monitors

Hardware in-circuit emulators using dual port ram (e.g., Lauterbach, AMC)



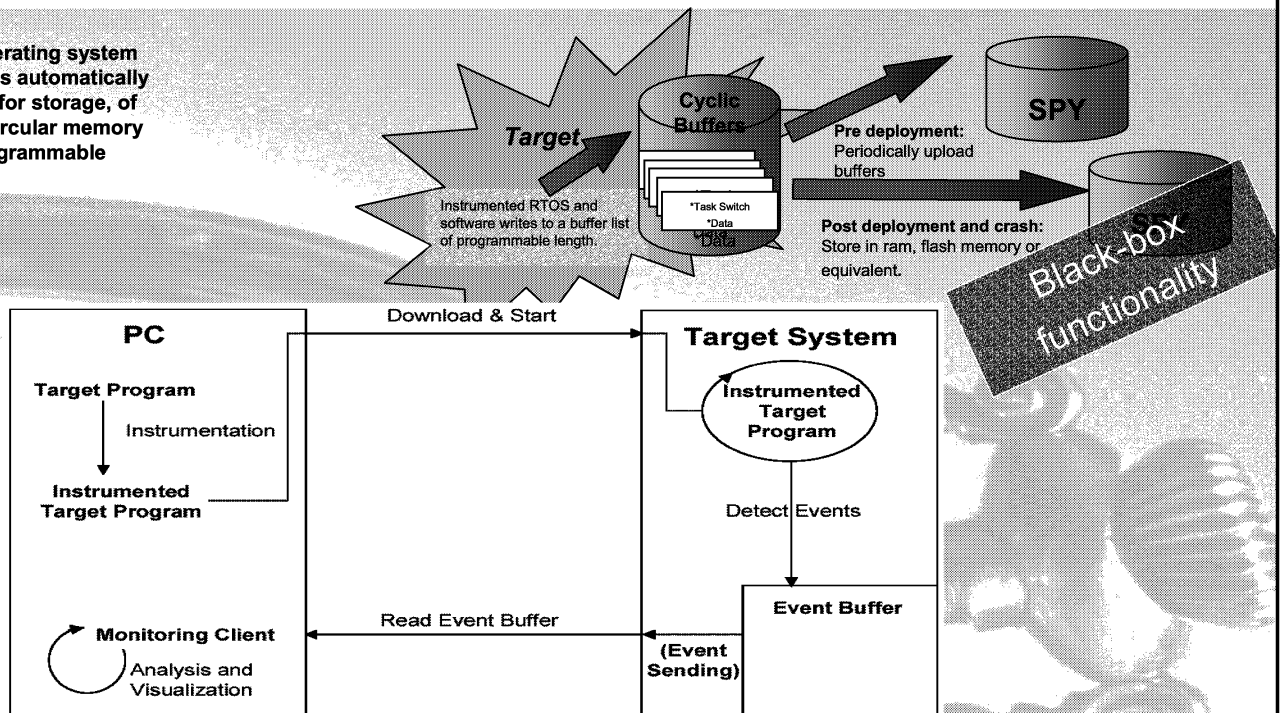
Non Intrusive Hardware Monitors



© Henrik Thane 2003 henrik.thane@mdh.se

Software Monitors

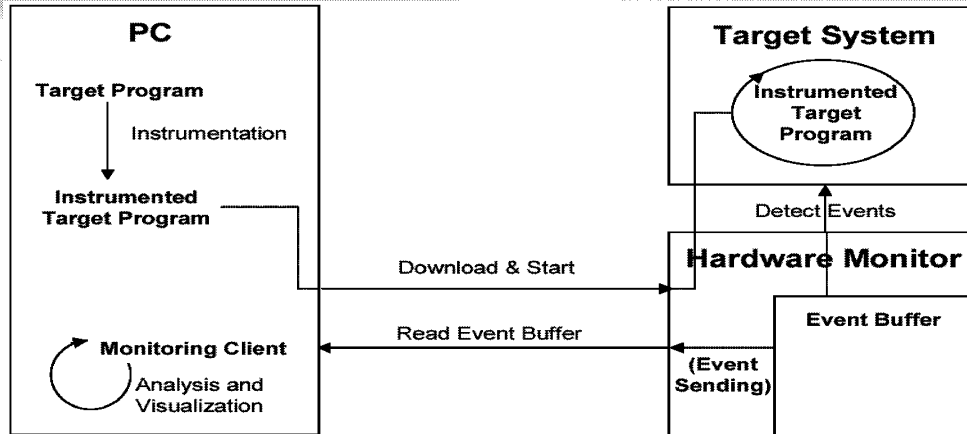
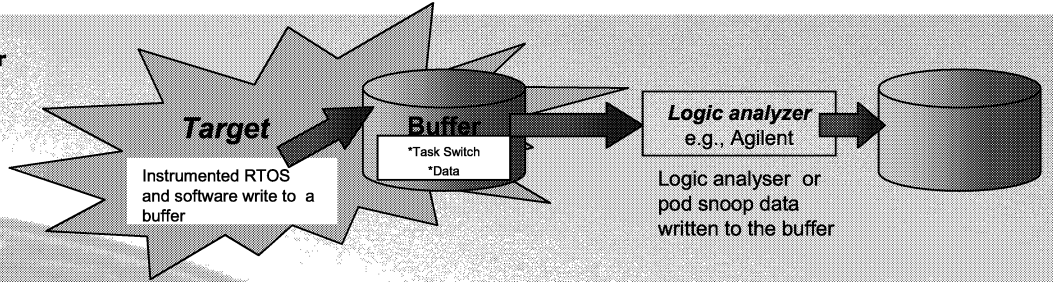
The target operating system and software is automatically instrumented for storage, of histories, in circular memory buffers of programmable length.



© Henrik Thane 2003 henrik.thane@mdh.se

Hybrid Hardware Software Monitors.

Hardware in-circuit emulators or logic analyzers collect histories using bus snooping and minimally instrumented software.



© Henrik Thane 2003 henrik.thane@mdh.se

Hybrid Monitor - Logic analyzer



© Henrik Thane 2003 henrik.thane@mdh.se

Software Monitoring

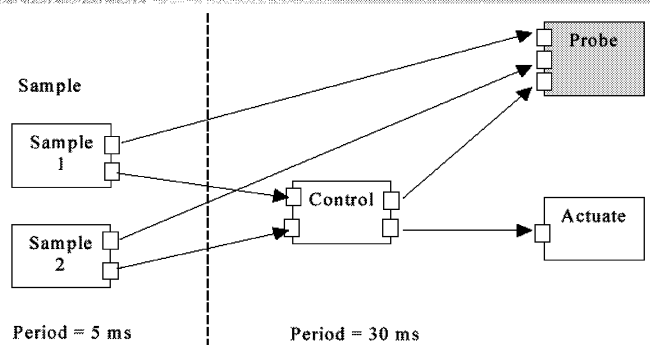
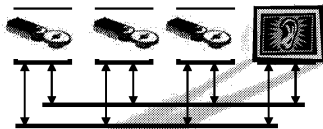
■ Intrusive observations

- **Solution:** Allocate resources for probes and leave them in the target system

■ Can observe on many levels

- Kernel Probes
- Inline probes
- Task probes
- Node probes

```
....
....
printf("red alert");
....
....
```



Design Rules for Minimal Intrusiveness

- **Minimize execution time jitter**
 - Invariant loops
 - Always max execution time
 - Equalize control flow
 - Use dummies
 - Atomicity
- **Minimize response time jitter**
 - Collection information locally in each task
 - Or, use low priority task

- **Group information**
 - Centralize input/output
 - Inter Process Communication
 - Environment I/O
 - Centralize internal state information
 - Reduce information
 - Try to design stateless functions if possible
 - Also improves reproducibility
- **On/Off facilities**
 - How to turn monitoring on/off
 - Zero Memory consumption → constant Exec Time.

Elimination of software monitors?

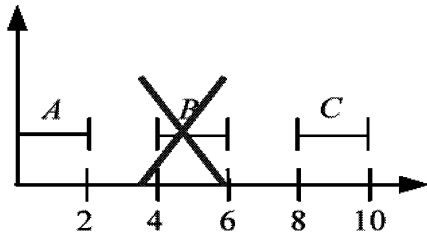


Figure 4-9. Probe task B can be removed due to fixed release times of A and C.

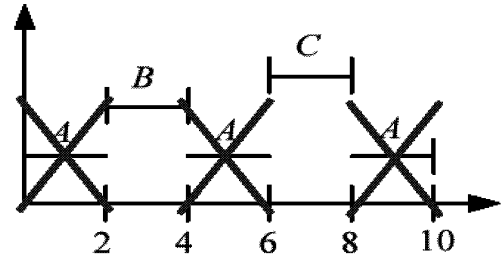


Figure 4-10. Low priority probe task A can be removed without side effects.

Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring, Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

Software testing for embedded systems

Basic stages

1. Module level testing
2. Integration testing
3. System testing
4. Hardware/Software Integration testing
 - This is unique to embedded systems.

Testing Concurrent Systems

- **Concurrency increases the difficulty of SW testing**
 - Unmanageably large set of legal execution sequences that a concurrent program may take.
 - The number of possible execution histories arising from the different interleavings of actions is enormous
 - Subsequent execution could lead to different-yet correct results.
 - Each execution of the program may give a different history so the behavior is not reproducible
- **Non-intrusive testing**
 - Embedded applications have strict timing requirements. Absolutely imperative that there be no intrusions on a test execution.

Embedded Testing

Critical issues

- Typically developed on custom HW configurations, each would require own set of tools and techniques.
- Errors discovered during H/S integration testing are most difficult of all. Often require significant modifications to the SW system.
- 2 environments
 - Host and the Target. Target has little support for SW development tools.

Problems with Embedded Testing

1. Expense of testing process
 - *Little reuse, expensive custom validation facilities required for every project. Retests extremely costly.*
2. Level of functionality on target
 - *Very low, hence a lot of effort to discover errors.*
3. Late discovery of errors
 - *SW modified to rectify HW errors, delays error discovery.*
4. Poor test selection criteria
 - *Test case selection rarely on theoretical criteria*

The Controllability and Determinism problem

Test of sequential software

- **Need only control sequence of inputs (easy :-)**

Test of multi-tasking real-time software

- **Need to control sequence of inputs**
- **Need to observe (control) timing**
- **Need to observe (control) order of execution**
- **Need to eliminate probe effects**

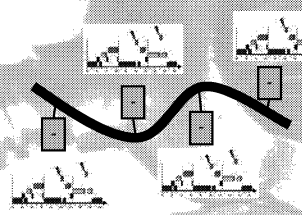
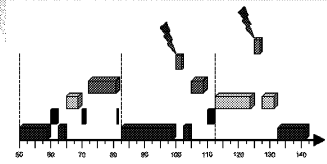
Test of distributed real-time systems software

- **Need also correlate observations between nodes**

Input

Program

Output



The Controllability and Determinism problem

Task A

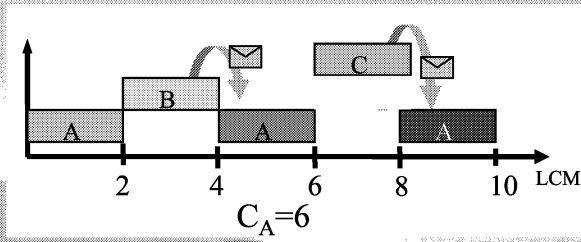
1. Sample (&X);
 2. Receive(task_B, &Y);
 3. Receive(task_C &Z);
- Output(X+Y-Z);

Task B

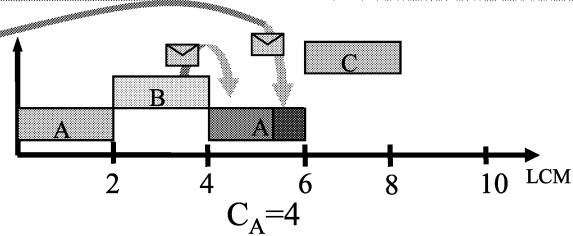
- Sample (&value);
- Send(task_A, value);

Task C

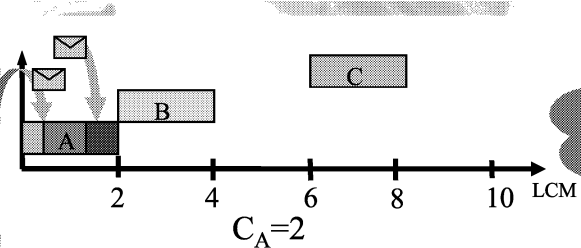
- Sample (&value2);
- Send(task_A, value2);



$Output(x+y_{new}-z_{new})$



$Output(x+y_{new}-z_{old})$



$Output(x+y_{old}-z_{old})$

Task A

1. Sample (&X);
2. Receive(task_B);
3. Receive(task_C);
- Output(X+Y-z)

Task B

Sample (&value);

Send(task_A, value);

$C_A=2$

LCM

Output(x+y_{new}-z_{new})

The same scenario → Deterministic output

Task B

Sample (&value2);

Send(task_A, value2);

$C_A=2$

LCM

Output(x+y_{old}-z_{old})

© Henrik Thane 2003 henrik.thane@mdh.se

A viable approach

Sequential prgs.

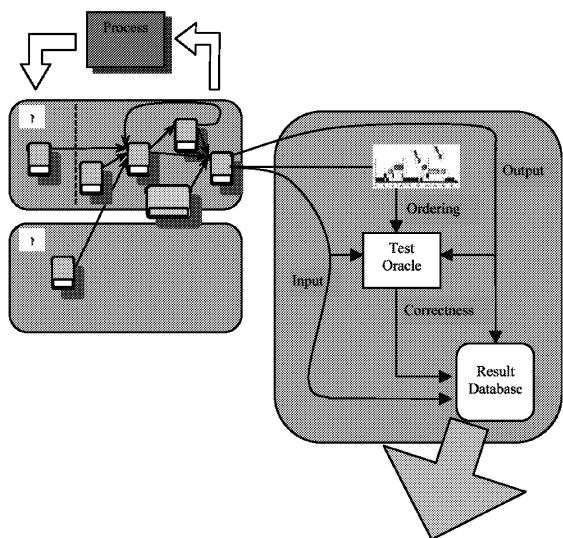
System level control-flow

Similarly for multitasking RTS

Each ordering can be regarded as a sequential program

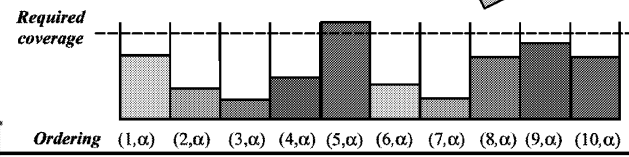
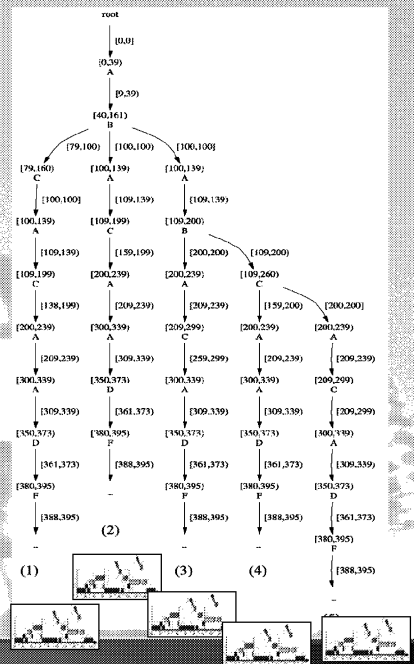
© Henrik Thane 2003 henrik.thane@mdh.se

A Deterministic Testing Method



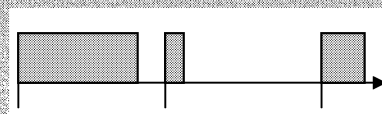
- Procedure**
1. Test system
 2. Observe path
 3. Store test result in path container
 4. Repeat 1-3 until sufficient no paths been covered
 5. Repeat 1-4 until sufficient coverage for each path been covered

- System control-flow testing levels**
- One task
 - One task + interrupts
 - One Transaction (several tasks)
 - Multiple transactions
 - All tasks
 - All nodes

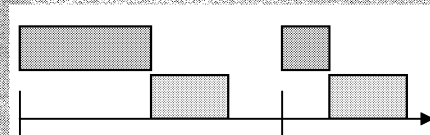


But when do you stop Testing???

We need coverage criteria!!
 How many system control flow paths are there?
 Depends on critical factors
 - Execution time jitter



- Start jitter
- Interrupt Jitter
- Communication time jitter
- Clock synchronization jitter



Approach

- Measure (statistically) the execution times of the tasks.
- Derive (using simulation or analysis) the possible paths.

Improving testability

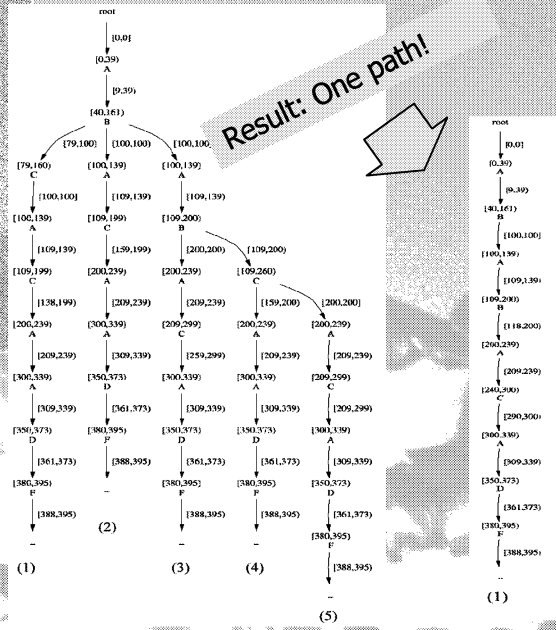
No. Paths = function(jitter):

- Execution time jitter
- Start jitter
- Clock synchronization jitter
- Communication latency jitter
- Interrupt induced jitter
- No. of synchronization points / preemption points

$$\approx 3^{n \cdot p}$$

Approach e.g.:

- Give fixed release times
- Reduce the execution time jitter for the tasks and critical sections



Controlled Testing

- Add random delays to synchronization points
 - Typically blocking IPC
 - Semaphore accesses
 - Task delays
- Record Reference executions
 - Replay control-flow and change inputs
 - Possibility of non valid executions

Outline

- Introduction
- Testing in general
- Testing of components and architectures
 - Reuse & test effort
- Design for high testability
- Monitoring, Testing and Debugging of RTS
 - Introduction
 - Monitoring
 - Testing
 - Debugging
 - Design for Testability

The Debugging Problem

Have you ever had problems during integration testing and debugging

- **With intricate timing problems?**
- **Hard to reproduce failures?**
- **Sporadic production stops after deployment of your embedded application?**

Debugging with Monitoring

Approaches

- Real-time display
 - display the system state in real-time
- Real-time debugging
 - extension to real-time display
 - plus: modification capabilities
 - but: rescheduling maybe necessary
- Interactive debugging
 - often as an extension to real-time debugging
 - plus: execution of the target program is suspendable and resumable on user request

Debugging with Monitoring 2

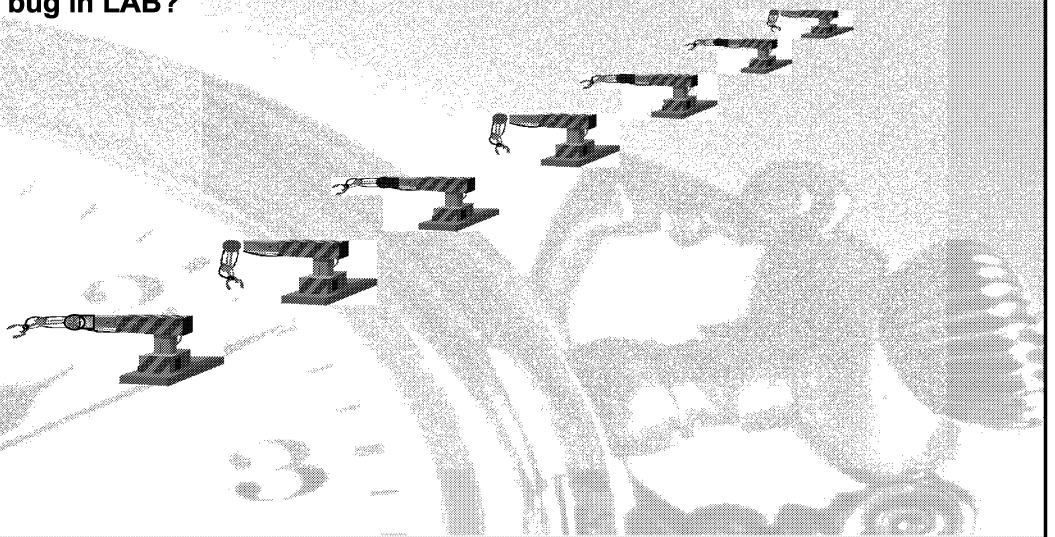
Approaches (cont.)

- Deterministic replay
 - store the detailed monitoring log
 - simulate execution based on the log
- Dynamic simulation
 - extension to deterministic replay
 - limited modification capabilities based on the monitoring log
 - but: simulation does not match the real world 100%
-

Debugging in Reality

Robots in the process industry:

- ▶ Debugging in prod. environment not possible
- ▶ Detection of the correct bug in LAB?



© Henrik Thane 2003 henrik.thane@mdh.se

The Debugging Problem

```
File Edit Search View Project Debug Tool Options Window Help
c:\vestide.c
void eog(
    node_t* n,
    job_t* rdy set,
    time_t a, time_t b,
    time_t sl, time_t sr)
{
    time_t t, alfa, beta, a_prime, b_prime, A, B, w;
    job_t *rdy, *T;
    node_t* n_prime;
    long call;

    recur++;
    call = recur;
    rdy = copy set(rdy set);
    t = next_release(sl, LCM);

#ifdef TRACE
    print_trace(n, rdy, a, b, sl, sr);
#endif

    if (rdy == NULL)
    {
        if ((rdy=make_ready(t, rdy)) != NULL)
        {
            eog(n, rdy, a, b, EXCL(t), sr);
        }
        else
        {
            Arc(n, a, b, NULL, call);
            orderings++;
        }
    }
    else
    {
        T = X(srdy);
        A = a < 0? EXCL(a): a;
        B = b < 0? EXCL(b): b;

        alfa = T->r > A? T->r: A;
        beta = (T->r > B? T->r: B) + w_max(T->wset);
    }
}
```

Cyclic debugging of sequential software (Easy :-)

- Implicit reproducibility
- Breakpoints,
- Watches,
- Traces (single stepping).

Input

Program

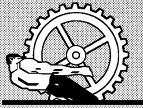
Output

© Henrik Thane 2003 henrik.thane@mdh.se

The Debugging Problem

Cyclic debugging of sequential real-time software

- Problem of timely reproduction of inputs/outputs.
 - How to breakpoint the world???!!
- The probe effect.



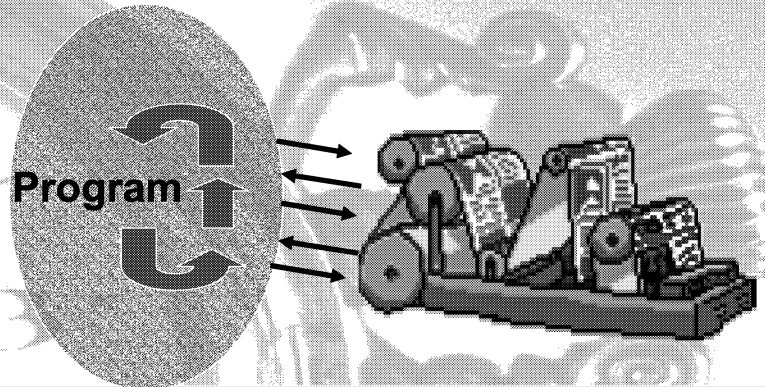
```

void eog( node_t* n,
         job_t* rdy_set,
         time_t a, time_t b,
         time_t s1, time_t sr)
{
    time_t t, alda, beta, a_prime, b_prime, A, B;
    job_t *rdy;
    node_t* n_prime;
    long call;

    recur++;
    call = recur;
    rdy = copy set(rdy_set);

#ifdef _TRACE
    print_trace(a, rdy, a, b, s1, sr);
#endif

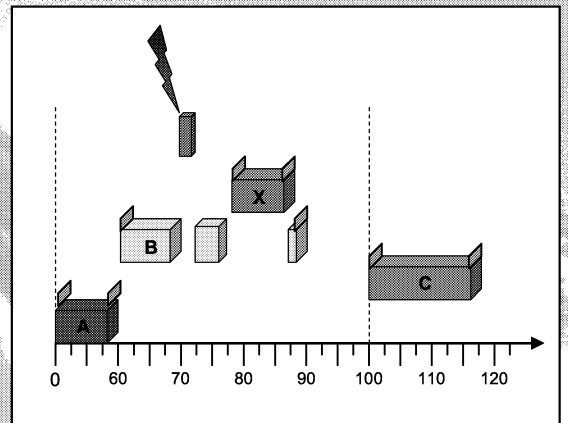
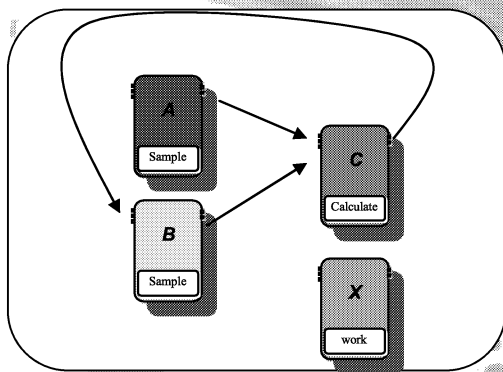
    if (rdy == NULL)
    {
        if (!rdy_make_ready(t, rdy) != NULL)
        {
            eog(n, rdy, a, b, EXCL(t), sr);
        }
        else
        {
            Arc(n, a, b, NULL, call);
            orderings++;
        }
    }
    else
    {
        T = X(rdy);
        A = a < 0? EXCL(a): a;
        B = b < 0? EXCL(b): b;
        alda = T->e > A? T->e: A;
        beta = (T->e > B? T->e: B) + v_max(T->wset);
    }
}
    
```



The Debugging Problem

Cyclic debugging of multi-tasking real-time software

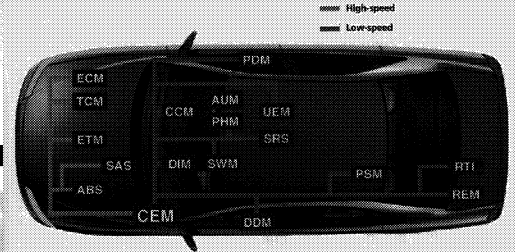
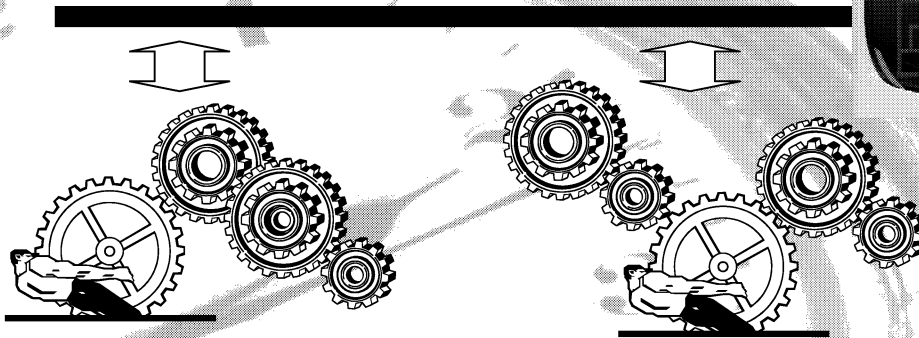
- Additional problem of reproducing task interleavings



The Debugging Problem

Cyclic debugging distributed real-time systems

- **Distributed breakpoints?**
- **Need also correlate observations between nodes**



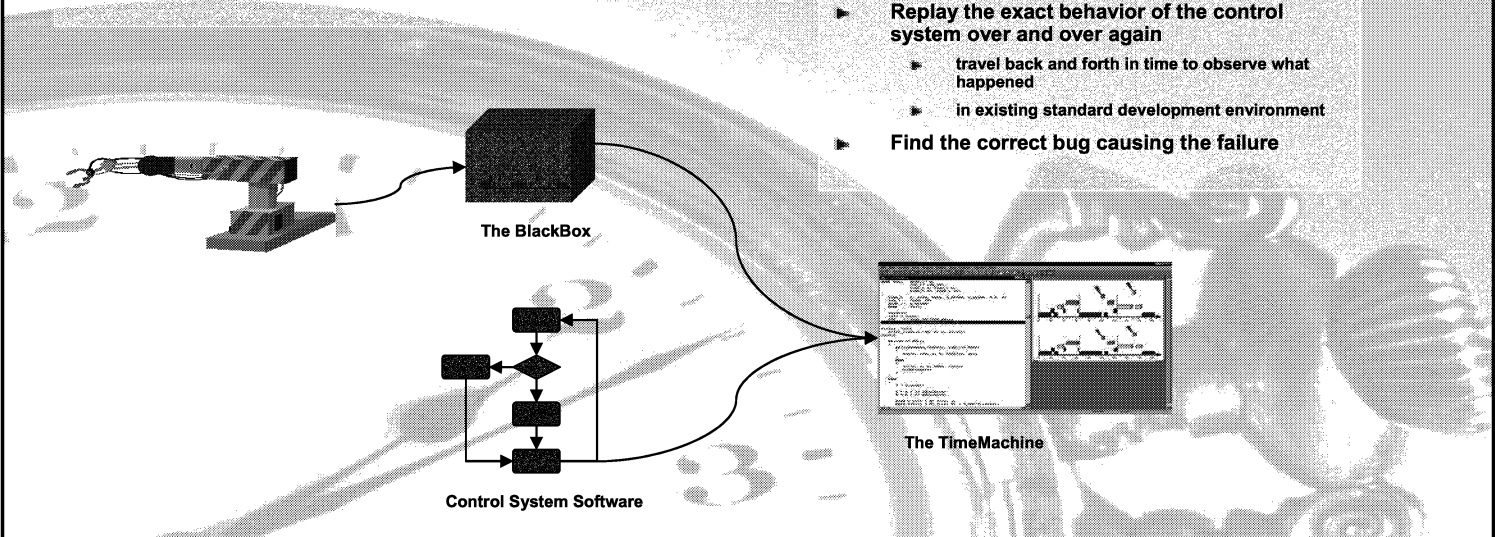
A Solution

Software "black-box" Recorders:

- ▶ Records behavior and stimuli during operation
 - Compared to traditional "black boxes"
- ▶ Minimal resource requirements

Deterministic Reexecution:

- ▶ Reexecutes the target system exactly according to the recording
 - mixes recording of real behavior with reexecution of control system software
- ▶ Replay the exact behavior of the control system over and over again
 - travel back and forth in time to observe what happened
 - in existing standard development environment
- ▶ Find the correct bug causing the failure



Deterministic Replay Debugging of RTS

Related work

Concurrent Systems

- *Reproducing rendezvous between tasks in ADA - Tai et al. 1991*
- *Choi J. D., et al. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications, 2001*
- *Zambonelli and Netzer. An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications, 1999*

Problem: Handle only synchronous events like rendezvous

Real-Time Systems

Reproduction of interrupts and task-switches using special hardware - *Tsai et al. 1990*

Reproducing interrupts and task switches using both special hardware and software - *Dodd and Ravishankar 1992.*

Problem: Relying on special hardware and no support for distribution.

Distributed Real-Time Systems

• *Thane H. and Hansson H. Using Deterministic Replay for debugging of Distributed Real-Time Systems. 2000.*

• **Problem:** Specialized real-time kernel

Most References are old > 10 years

Not much progress lately!

No Industrial Application

- ***The Idea has been around for 20 years***
 - **Still no Industrial application**
 - **Previous solutions do not scale**
 - Require special hardware, special compilers, special RTOSs, or significant application code modification
 - No support for Standard Components only academic projects
 - Cannot handle large code volumes/ many tasks (MLOC)
- **A Deterministic Replay Technology Necessary**
 - Based on Standard Commercial RTOSs and Debuggers
 - Must work with existing application code base

Our Approach

Define the industrial problem

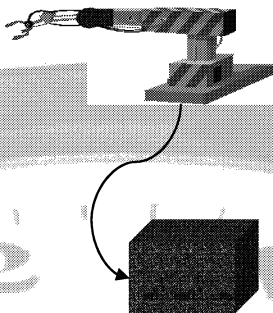
- ▶ Assume standard commercial RTOS
 - ▶ E.g., VxWorks (30% market share)
- ▶ Assume interface to application via standard interactive debugger
 - ▶ Language independent
- ▶ Assume multi-million lines of code
 - ▶ Cannot manually instrument source
- ▶ Recording performance penalty must be low (and constant)

Identified success factors

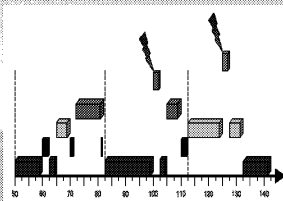
- ▶ Use existing "hooks" in RTOSs
- ▶ No reliance on instruction counters
 - ▶ Cannot be used with standard RTOSs
- ▶ Use context checksums based on accessible task context for markers
- ▶ Control Application and RTOS via breakpoints in debugger
 - ▶ Abstracts away the target hardware and language

Our Solution

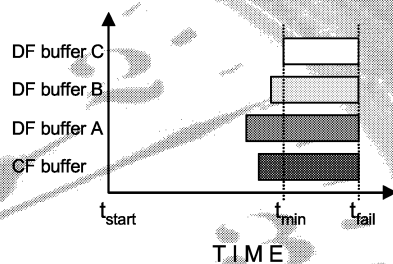
The Recorder



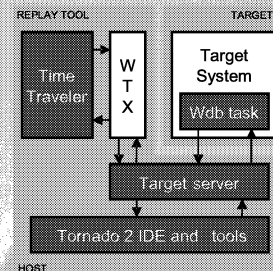
The Historian



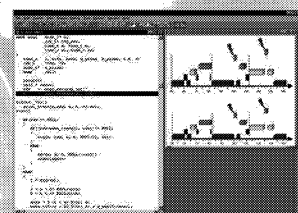
- ▶ Identify possible starting points



The Time Traveller



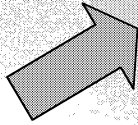
- ▶ Set breakpoints at recorded control-flow events



Our Solution

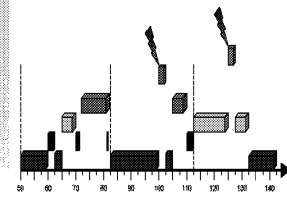
▶ The Recorder

- ▶ Control-Flow:
 - ▶ Asynchronous Preemptions
 - ▶ Scheduled Task-Switches
 - ▶ Interrupts
 - ▶ Synchronous Preemptions,
 - ▶ Blocking system calls
- ▶ Data-flow:
 - ▶ Selective Inter-process communication
 - ▶ External non-deterministic stimuli
 - ▶ Task state
- ▶ Use Hardware, Software or hybrid recorders



▶ The Historian

- ▶ Correlate Data-Flow with Control-Flow and generate Time Line

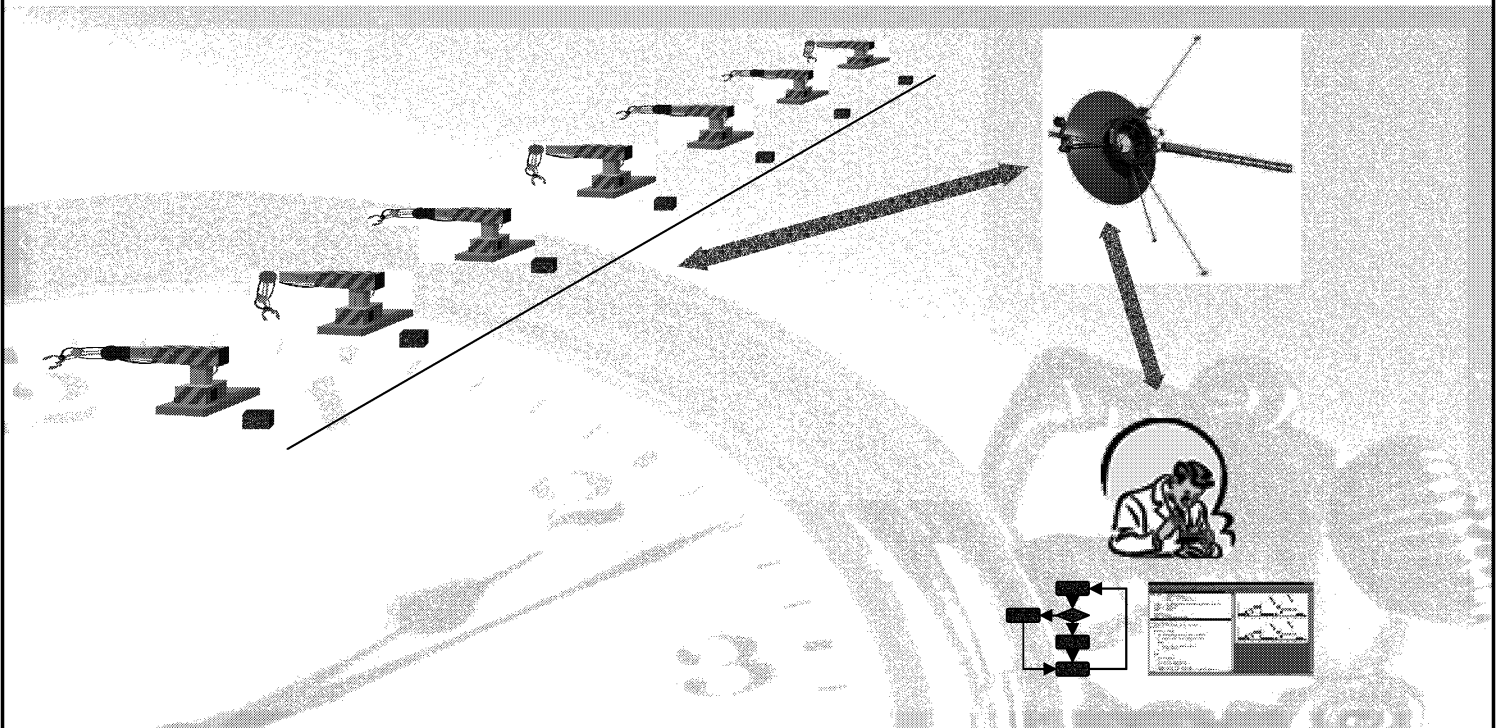


- ▶ Generate breakpoints according to timeline

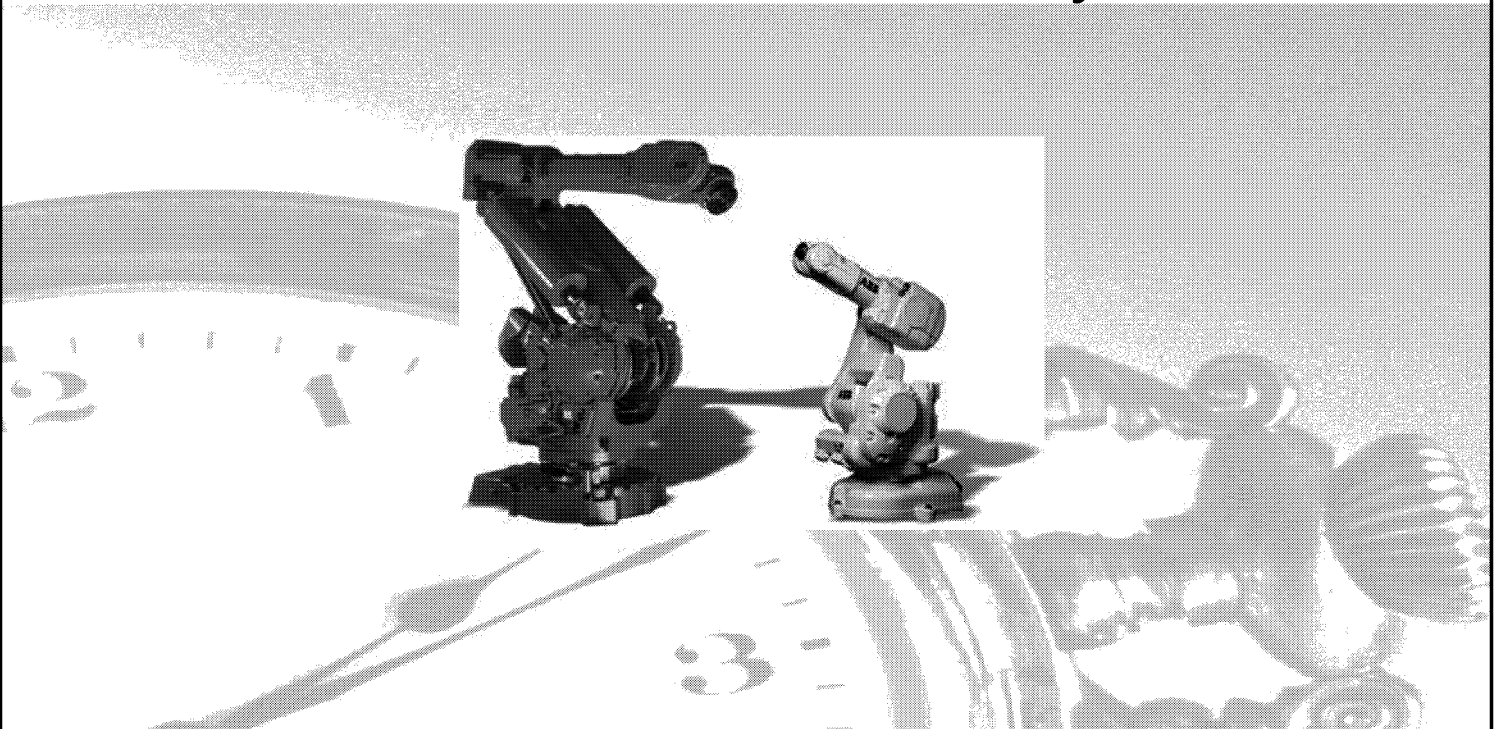
▶ The Time Traveller

- ▶ Forces the target system to the same state as stored in the recording
- ▶ React on breakpoint hits
- ▶ Transform state

Black-Box Recorders



Industrial Case-Study



© Henrik Thane 2003 henrik.thane@mdh.se

Motion Control Software

The image shows a rack-mounted industrial computer system with several components labeled with boxes and lines pointing to them:

- I/O computer
- Axis computer
- Main computer
- PCI Backplane
- Flash disk
- Cooling fan
- Computer power supply
- Battery unit

Motion Control Software

- 70 tasks
- Most frequent task: $T=4\text{ms}$
- Plenty of interrupts
- 2.5 Million LOC
- Hard Real-Time System

RTOS

- VxWorks

IDE

- Tornado

© Henrik Thane 2003 henrik.thane@mdh.se

Implementation

- **Black-Box/Monitoring infrastructure for VxWorks**
 - Task switches
 - System calls
 - Message queues
 - Up/download
- **Black-Box/Monitoring infrastructure for Application**
 - Data structure e.g., state
- **Replay tool / Time Machine**
 - Preemption, semaphores, ipc, task delay, data state

Integration

With Tornado via WTX

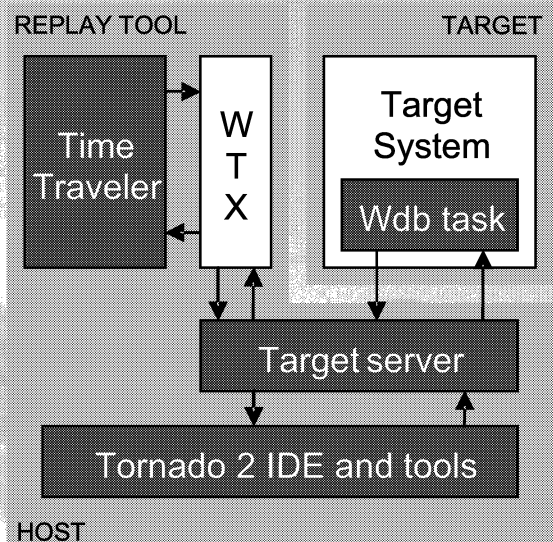
Replay engine
Target server
Debugger (Task mode)

MC Robot application

Monitoring

- Task switches
- System calls
- IPC
- Data structures

Execution time and bandwidth
measurements



Black-Box Overhead

Nr of Bytes/function		CPU-cycles	Time (approx.)	ΣC/T For all tasks	< 3% μP utilization
State variable monitoring	360 bytes	1992	10 μs		
	1024 bytes	3797	18 μs		
	8192 bytes	18545	93 μs		
IPC	360 bytes	~1000	5 μs		= 0,05 % μP utilization
Function calls	semTake, msgQReceive, etc.	~30	< 0,2 μs		
Taskswitch (Interrupts disabled)		378	1,9 μs		

Data-flow bandwidth 2MB/s.
Interrupts not disabled, i.e., interrupt overhead not deducted. Non-optimized code (neither algorithm nor compiler)

Additional benefits with TM & Black-Boxes

- Use it for controlled testing
 - Vary recorded data-flow with fixed control-flow
- Do high-level Debugging on optimized code
 - Concurrency control-flow recorded
 - Use code with debug info off-line, record with no debug info on-line.
- Reduced diagnostics load
 - System state is recreated through reexecution
- Eliminated repetitive compile-link-load cycles

Conclusion

- **Testing software is fundamentally hard**
 - **Corrective cost of software is significant (more than 40%)**
 - **Corrective cost of embedded software (40%- 80%) [NIST]**
 - **Brute force is not a viable approach in the long run**
- **Design for high testability essential**
 - **Increase the observability**
 - **Assertions, BIST**
 - **Domain to range ratio (increase the output)**
 - **There exist a fundamental tradeoff between fault tolerance and testability**
- **Reuse of software does not necessarily mean "no" verification rather the opposite**
 - **Incrementally build assertions and BIST**
- **Testing will not find specification errors**

Conclusion (continued)

- **Proper diagnostics fundamental for successful debugging and testing of embedded software**
 - **Need to consider probe-effect issues**
 - **Low jitter design**
 - **Centralized storage**
- **Testing of Multi-tasking software need to address**
 - **Determinism and reproducibility issues**
 - **Many legal execution scenarios**
 - **Design for high testability**
 - **= Low jitter design**
 - **= Few synchronization points**
 - **= few data access points**
 - **= Stateless functionality If possible**
- **Debugging of Multi-tasking software need to address**
 - **Elimination of the probe-effect**
 - **Use Deterministic Replay and Black-Box functionality**

References

- [SEN1] ACM Software Engineering Notes , Vol.8, No. 3.
- [DeMarco78] T DeMARco. Structured Anlysis and System Specification. Yourdon Press 1978. ISBN 0-917072-07-3
- [Ell95] A. Ellis. Achieving Safety in Complex Control Systems. Proceedings of the Safety-Critical Systems Symposium, Brighton, England, 1995. Springer-Verlag. ISBN 3-540-19922-5
- [NIST] NIST Report. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. U.S. Department of Commerce. May 2002.

Sponsored by:



Alan Shaw

University of Washington

THE TIMING BEHAVIOR OF EMBEDDED SYSTEMS:
SPECIFICATION, PREDICTION, AND CHECKING

Alan Shaw

1. Introduction *
 - Overview (1.1, 1.1.1, pp.1-4)
 - Applications of time and clocks (8.1, pp.136-137;
2.1, 2.1.1, pp.15-18)
2. Keeping Time on Computers
 - Properties of clocks (8.2, pp.137-139)
 - Time servers (8.3, pp.139-140; 8.3.2, pp.142-143)
 - Clock synchronization (8.4, pp.144-149)
3. Specifications with Time
 - Imperative and declarative methods (3.1, pp.33-35; 5, p.76)
 - State machines (3.4, pp.45-49)
 - Communicating real-time state machines (4.1, pp.50-65)
 - Regular expressions and extensions (5.1, 77-81)
 - Real-time logic (5.3, pp. 87-90)
4. Predicting Program Execution Times
 - Issues and approaches in timing prediction (7.1, pp.110-114)
 - Measurement techniques (7.2, pp.118-129)
 - Program analysis with timing schema (7.3, pp.118-129)
 - Software and architectural complexities (7.5, pp.134-135)
 - Prediction using optimization methods (7.4, pp.130-133)
5. Software Support for Time
 - Programming language features (9.1, pp.150-151)
 - Ada as an embedded systems language (9.2, pp.151-162)
 - Java and real-time Java (9.4, pp.168-171)
 - Operating systems support (10.1, pp.182-183;
10.2.1. pp.184-186; 10.3.3, pp.190-192)

* The parenthesized parts refer to section and page numbers in the book:
Alan C. Shaw, "Real-Time Systems and Software",
John Wiley & Sons, Inc., 2001.

L1

INTRODUCTION

* Definitions: Embedded and Real-Time Systems

* Applications of Time and Clocks

KEEPING TIME ON COMPUTERS

* Properties of Real and Ideal Clocks

* Clock Servers

* Clock Synchronization

 With a Centralized Standard

 Distributed Internal Synchronization: Averaging Algorithm

L2.1

SPECIFICATIONS WITH TIME

* Imperative vs Declarative Methods

* State Machines

Finite State Machines

Extended Machines

* Communicating Real-Time State Machines

Basic Distributed Machines

Timing and Clocks

Examples: Alarm Clock, Mouse Clicker, Philips Defibrillator

Semantics, Tools, and Extensions

SPECIFICATIONS WITH TIME (cont'd)

* Regular Expressions and Extensions

Regular Expressions and Finite State Machines

Extensions for Concurrency

* Real-Time Logic

Predicate Logic with Events

Event Occurrence Function

Application to Run-Time Checking

PREDICTING PROGRAM EXECUTION TIMES

* Approaches and Issues

Reasoning about Time with Assertions

Measurement vs Simulation vs Analysis

Underlying Hardware and Software

* Measurement Techniques

* Program Analysis with Timing Schema

Concepts: Static Analysis

Dynamic Path Analysis with Extended Regular Expressions

Schema Practice: Tools and Experiments

* Prediction by Optimization

Integer Linear Programming Approach

* System Interferences and Hardware Complexities

SOFTWARE SUPPORT FOR TIME

* Programming Language Features

* Ada as an Embedded Systems Language

Core Facilities: Concurrency, Exceptions

Real-Time Annex

Interrupt Model

* Java and Real-Time Java

Real-Time Threads

Timing Mechanisms

* Operating Systems Support

Real-Time Features

Real-Time UNIX and POSIX

Synchronization and Communications

Dr. Hyuk Jae Lee

Seoul National University

Features and Internals of Embedded Linux for Real- Time Operations

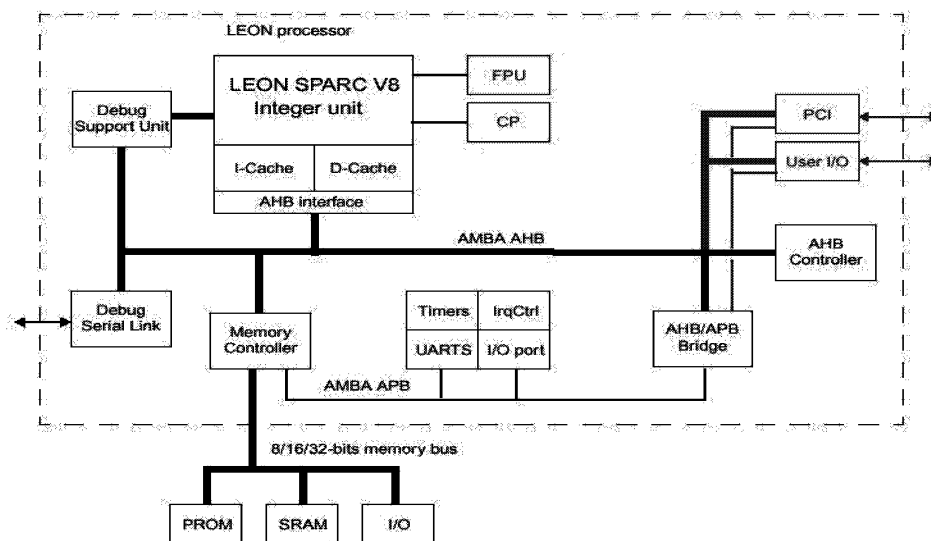
Hyuk-Jae Lee
Seoul National University

Contents

- Embedded Systems and Device Driver Programming
- Embedded SW Architecture
- Linux Device Driver Interface
- Linux Features
- Linux Kernel Internals
- Real-time scheduling
- Linux Device Driver Programming
- Block Driver Interface
- Real-time Synchronization

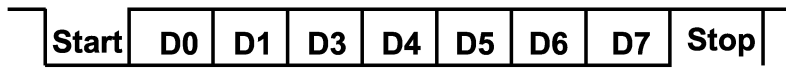
Embedded Systems and Device Driver Programming

Leon: An Embedded System

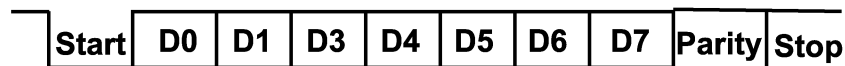


Peripheral Example: UART

- Data frames with 8 data bits, one optional parity bit, one stop bit.

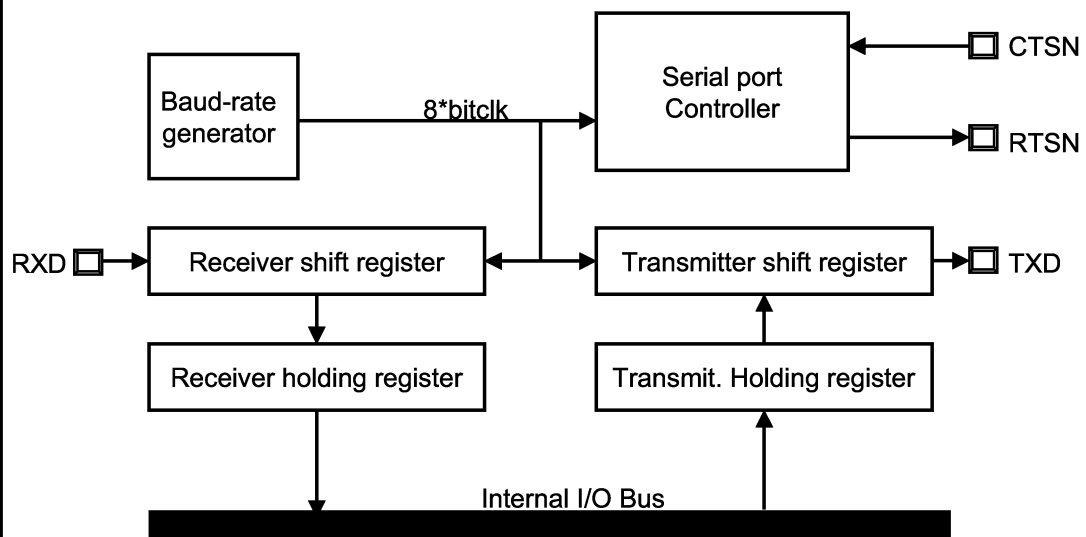


Data frame, no parity



Data frame with parity

UART: Block Diagram



UART Operation

- Transmitter:
 - Enabled through the TE bit in the control register.
 - Data is transferred from the transmitter holding register to the transmitter shift register
 - Converted to a serial stream on the output pin (TXD)
- Receiver:
 - Enabled thru the RE bit in the control register
 - Look for a start bit (high to low transition)
 - Data transferred to the receiver holding register (RHR)
 - Data ready (DR) bit is set in the status register
 - Overrun error: receiver holding & shift registers contain an un-read character when a new start bit detected

Control and Status Registers

- Basic interface between processor and peripheral
- Part of the peripheral hardware
- Locations, size, and individual meanings are features of the peripheral.
- Memory-mapped: located in the processor's memory space
 - Easier to work with and are increasingly popular.
- I/O mapped: located in the processor's memory space
- Memory mapped peripherals are generally easier to work with and are increasingly popular.

UART Control Register

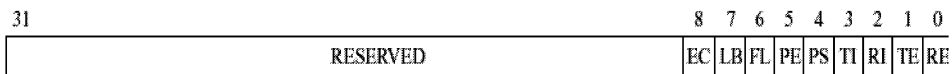


Figure 25: UART control register

- 0: Receiver enable (RE) - if set, enables the receiver.
- 1: Transmitter enable (TE) - if set, enables the transmitter.
- 2: Receiver interrupt enable (RI) - if set, enables generation of receiver interrupt.
- 3: Transmitter interrupt enable (TI) - if set, enables generation of transmitter interrupt.
- 4: Parity select (PS) - selects parity polarity (0 = odd parity, 1 = even parity)
- 5: Parity enable (PE) - if set, enables parity generation and checking.
- 6: Flow control (FL) - if set, enables flow control using CTS/RTS.
- 7: Loop back (LB) - if set, loop back mode will be enabled.
- 8: External Clock - if set, the UART scaler will be clocked by PIO[3]

UART Status Register

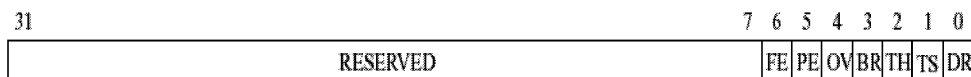


Figure 26: UART status register

- 0: Data ready (DR) - indicates that new data is available in the receiver holding register.
- 1: Transmitter shift register empty (TS) - indicates that the transmitter shift register is empty.
- 2: Transmitter hold register empty (TH) - indicates that the transmitter hold register is empty.
- 3: Break received (BR) - indicates that a BREAK has been received.
- 4: Overrun (OV) - indicates that one or more character have been lost due to overrun.
- 5: Parity error (PE) - indicates that a parity error was detected.
- 6: Framing error (FE) - indicates that a framing error was detected.

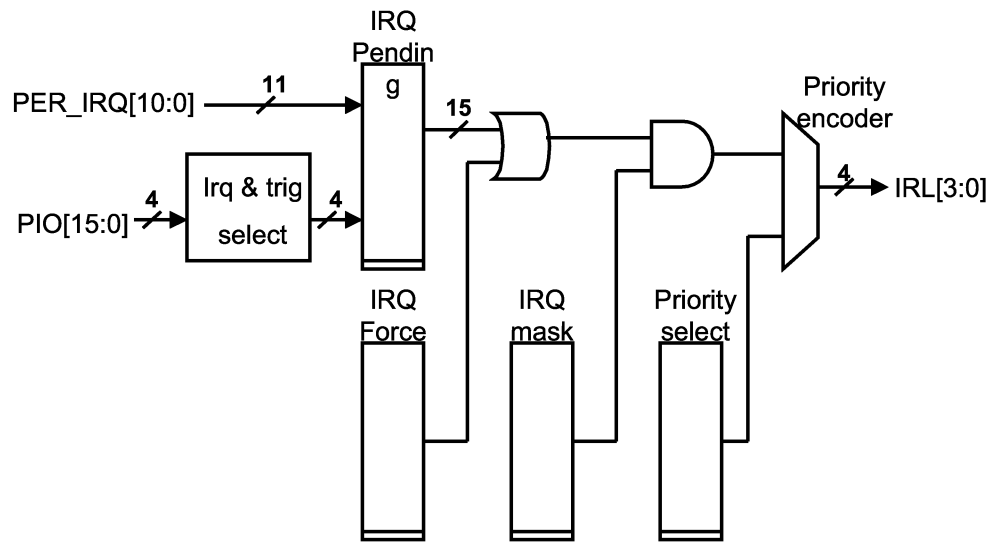
Device Driver

- Main function: access the control and status registers
- Goal: hide the hardware
 - The only piece of software that reads or writes that particular device's registers.
 - Device driver programming interface would not need to be changed even if the peripheral were replaced with another
- Memory-mapped registers look just like ordinary variables.
 - Declare a pointer to the register/set the value of the pointer
 - For example, Timer2 Count Register at 0x80000050
`unsigned int* timer2CntReg = (unsigned int*)0x80000050`
`*timer2CntReg ^= 0xFFFFFFFF; // Read, XOR, Modify`
- Difference between registers and ordinary variables:
 - The content may change → use volatile
`volatile unsigned int* timer2CntReg = (unsigned int*)0x80000050`

Interrupt

- Response time issue:
 - The embedded system needs to react rapidly to external events, even in the middle of doing something else.
- Interrupts:
 - Microprocessor to suspend the current job
 - Execute some different code (interrupt service routine) to respond to whatever event caused the interrupt.
- Peripheral operations: need help from microprocessor
 - For example, when a serial port chip receives a character, it needs the microprocessor to read that character from the serial port chip itself and to store it somewhere in memory.
 - Peripheral has a pin that it asserts when it requires service.
 - This pin is attached to an input pin on the microprocessor called an interrupt request, or IRQ, that lets the microprocessor know that some other chip in the circuit wants help.

Leon Interrupt Control Block



Leon Interrupt Assignment

Interrupt	Source
15	user defined
14	user defined
13	user defined
12	user defined
11	DSU trace buffer
10	Second interrupt controller
9	Timer 2
8	Timer 1
7	Parallel I/O[3]
6	Parallel I/O[2]
5	Parallel I/O[1]
4	Parallel I/O[0]
3	UART 1
2	UART 2
1	AHB error

Interrupt Table

- Q: How does the microprocessor know where to find the interrupt routine when the interrupt occurs?
- A: Sparc has a Trap Base Register (TBR)

TBA	tt	zero
31:12	11:4	3:0

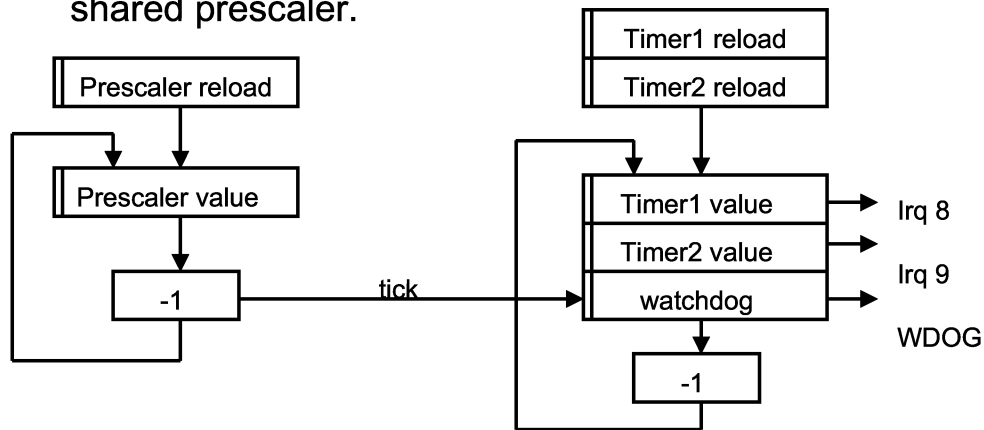
- TBA (trap base address): most-significant 20 bits of the trap table address
- tt (trap type): offset into the trap table; written by the hardware when a trap occurs.

Trap Table Offset

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Instruction or data watchpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception

Design of Timer Unit Driver

Two 24-bit timers, one 24-bit watchdog, one 10 bit shared prescaler.



Timer Unit Operation

- Prescaler
 - Decrement on each clock
 - When underflow
 - Timer tick generated
 - Reloaded from the prescaler reload register
 - Effective division rate: prescaler reload register+1
- Timer
 - Enabled by setting the enable bit (EN) in the control register
 - Decrement each time prescaler generates a timer tick
 - When underflow
 - Interrupt generated
 - if reload bit (RL) is set, reloaded with the timer reload register
 - Otherwise, stop at 0xfffff and reset the enable bit

Timer Unit Operation

- Watchdog
 - Similar to the timers
 - Always enabled
 - WDOG is generated when underflow
 - Used to generate a system reset
- Timer $\frac{1}{2}$ control register
 - LD: load counter
 - RL: Reload register
 - EN: Enable the timer

Data Structure

```
struct timerRegister
{
    unsigned int ctrl;
    unsigned int reload;
    unsigned int cnt;
};
struct timerRegister* timer2Reg = (struct timerRegister*) 0x80000050;
#define TIMER_ENABLE    0x0001
#define TIMER_LOAD_COUNTER 0x0004 // load the timer
                                // reload register
                                // into the timer counter register
#define TIMER_RELOAD_COUNTER 0x0002 // counter
                                // automatically reloaded
                                // w/ the reload value after underflow
#define TIMER2_IRQ_NUMBER 9
```

Variables

```
enum timerType { oneShot=1, periodic };
enum timerState { idle=1, active, done };

static unsigned int timerLength;
static unsigned int timerCount;
static unsigned int timerState;
static unsigned int timerType;
```

Initialization Routine

```
void timerInitialize() {
    // install interrupt handler
    catch_interrupt(timer2IrqHandler, TIMER2_IRQ_NUMBER);
    // make the hardware to a known state
    timerState = idle;
    timerType = oneShot;
    timerLength = 0;
    // enable Timer 2
    timer2Reg->reload = 0x01FFFF;
    timer2Reg->ctrl = TIMER_LOAD_COUNTER;
    // load the reload register value
    // to the counter register
    timer2Reg->ctrl = TIMER_RELOAD_COUNTER | TIMER_ENABLE;
    // enable Timer 2
    enable_irq(TIMER2_IRQ_NUMBER);
}
```

API Routines

```
#define MS_PER_TICK 10          // assume one milliseconds take 10
ticks

void timerStart(unsigned int nMilliseconds, unsigned int timerTypeParam)
{

    timerLength = nMilliseconds/MS_PER_TICK;
    timerType = timerTypeParam;
    timerCount = timerLength;
    timerState = active;

}
```

API Routines

```
int timerWaitFor()
{
    if (timerState != active) return (-1);

    while(timerState != done) {}

    if(timerType == periodic) {
        timerState = active;
        timerCount = timerLength;
    } else
        timerState = idle;

    return (0);
}
```

Interrupt Service Routines

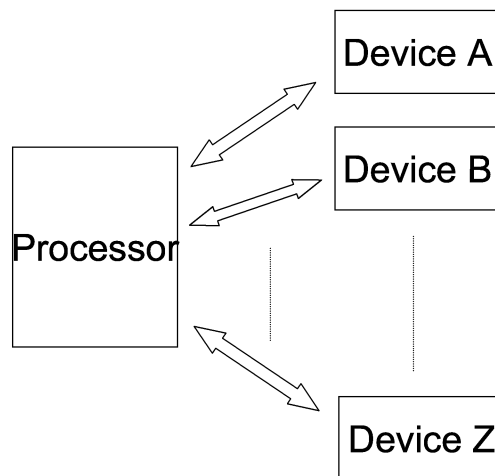
```
void timer2IrqHandler(int irq) {  
    timerCount--;  
    if(!timerCount) timerState=done;  
}
```

Application Program Example

```
main() {  
  
    timerInitialize();  
  
    timerStart(200, periodic);  
  
    while (1) {  
        timerWaitFor();  
        printf("Timer Expired !\n");  
    }  
}
```


Embedded SW Architecture

Support of Multiple Devices



Round-Robin

```
void main (void) {
    while (TRUE) {
        if (!! I/O device A needs service) {
            !! Take care of I/O device A
            !! Handle data to or from I/O device A
        }
        if (!! I/O device B needs service) {
            !! Take care of I/O device B
            !! Handle data to or from I/O device B
        }
        .....
        if (!! I/O device Z needs service) {
            !! Take care of I/O device Z
            !! Handle data to or from I/O device Z
        }
    }
}
```

Round-Robin

- Simple architecture: no interrupts, no shared data, no latency concerns
- Suitable device example: digital multimeter that measures electrical resistance, current, and potential
 - Only 3 I/O devices, no particularly length processing, no tight response requirements
- Disadvantages of round-robin
 - If a device needs response in less time than the microprocessor to get around the main loop in the worst scenario, then the system won't work.
 - The system may not work if any single lengthy processing exits.
 - This architecture is fragile. A single device may break everything.

Round-Robin w/ Interrupts

```
void main (void) {
    while (TRUE) {
        if (fDeviceA) {
            fDeviceA = FALSE;
            !! Handle data to or from I/O Device A
        }
        if (fDeviceB) {
            fDeviceB = FALSE;
            !! Handle data to or from I/O Device B
        }
        .....
        if (fDeviceZ) {
            fDeviceZ = FALSE;
            !! Handle data to or from I/O Device Z
        }
    }
}
```

Round-Robin w/ Interrupts

- Interrupt routines deal with the very urgent needs of the hardware and then set flags
- The main loop polls the flags and does any follow-up processing required by the interrupts.
- Interrupt routines get good response.
- Priority
 - Interrupt service routines always have higher priority than main function
- Disadvantage
 - Shared data problem: fDeviceA, fDeviceB, ... fDeviceZ

Function-Queue Scheduling

- Queue of functions: the pointers to functions to be executed.
- Interrupt routines add function pointers to the queue
- Main function calls the function in the queue
- Advantage:
 - Can give priority in the function call
 - Any function that needs quicker response can be executed earlier by putting the function on the queue in a given priority.
- Higher priority task can have reduced response time while lower priority task may have increased response time.
- If lower-priority task is long, the response time for the higher-priority task might be still quite long.
→ real-time operating system needed

Function-Queue-Scheduling

```
void interrupt vHandleDeviceA (void) {  
    !! Take care of I/O Device A  
    !! Put function_A on queue of function pointers  
}  
void interrupt vHandleDeviceB (void) {  
    !! Take care of I/O Device B  
    !! Put function_B on queue of function pointers  
}  
.....  
void interrupt vHandleDeviceZ (void) {  
    !! Take care of I/O Device Z  
    !! Put function_Z on queue of function pointers  
}
```

Function-Queue-Scheduling

```
void main (void) {
    while (TRUE) {
        while(!Queue of function pointers is empty)
            !! Call first function on queue
        }
    }
}

void function_A (void) {
    !! Handle actions required by device A
}

void function_B (void) {
    !! Handle actions required by device B
}

.....

void function_Z (void) {
    !! Handle actions required by device Z
}
```

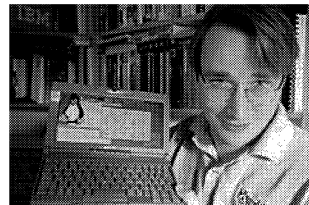
Real-time Operating System

- Can suspend one task code subroutine in the middle of its processing in order to another → Potential response time is zero.
- Issue:
 - When the currently running task needs to be suspended
 - The context of the suspended needs to be saved → additional overhead by the OS.
 - How to take care of the priority ?
 - At what point, OS needs to reschedule ?

Overview of Linux & Embedded Linux

Linux

- freely available
 - Linus Torvalds, Copyleft
 - 1991 version 0.01 (November 1999, version 2.2.13)
 - Redhat, Debian, Slackware, Alzza
 - supported many companies
- Main characteristics
 - multi-tasking
 - multi-user access
 - multi-processor
 - support various architecture (80*86, sparc, mips, alpha, smp, ..)
 - demand load executables
 - paging
 - dynamic cache for hard disk



Linux

- main characteristics (cont')
 - shared library
 - support for POSIX 1003.1
 - various formats for executable files
 - true 386 protected mode
 - emulating maths co-processor
 - support for national keyboards and fonts
 - support diverse file system (ext2, ..)
 - TCP/IP, SLIP, PPP
 - BSD sockets
 - System V IPC
 - Virtual Console

Linux

- drawbacks
 - monolithic kernel (currently micro kernelize in many research)
 - not for beginners (for system programmers)
 - not well structured (performance-oriented)
- **Key attraction**
 - 'experimenting' with the system (handle the kernel by yourself)
 - supported many companies
 - free: solution business & add on features
 - thanks to the INTERNET & GNU (special thanks to Anti-MS feeling)

Embedded Linux

- Embedded System
 - Those systems to control specialized hardware in which the computer system is installed
 - Hardware Evolution
 - Complex Software -> require OS
- Linux
 - Full-featured general OS
 - Portability
 - Kernel architecture
 - Open source
 - Relatively small kernel
 - Cost effective

Embedded Linux - Pros

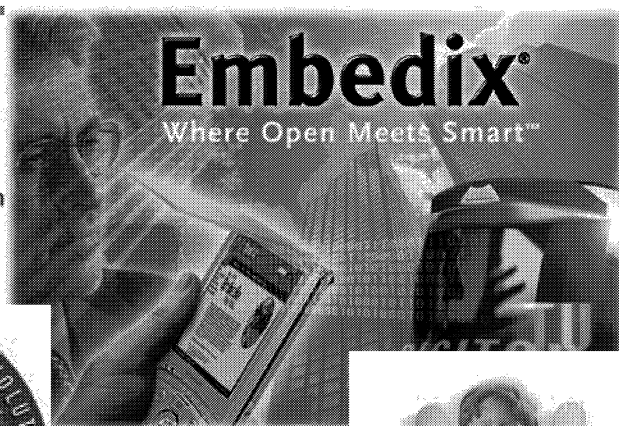
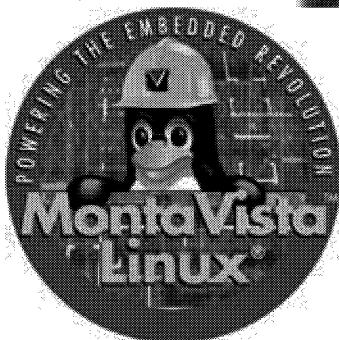
- A variety of functions & Expandability
- Multiple CPU platforms support
- Cost effective
 - No royalty
- Stability
 - Fast bug fix
- Easy application development
 - Same development environment as Desktop and Server
- Multi-Vendor OS
 - Independent from vendors

Embedded Linux - Cons

- Size
 - Smaller than General OS, but bigger than commercial RTOS
- Real-time Scheduling
 - Research efforts on Real-Time Scheduler
 - Limited preemption yet, (not fully preemptible)
- Lack of Integrated Development Environment
 - In general, text based development environment
- Difficult to start with
 - Too many open source sites and vendors

Embedded Linux Packages

- ◆ Collinux (Collogic)
 - <http://www.coollogic.com>
- ◆ XLinux (Coventive)
 - <http://www.coventive.com>
- ◆ RedBlue Linux (sfia)
 - <http://www.esfia.com>



- <http://>
- ◆ Embedded
 - <http://>
- ◆ ETLinux
 - <http://>



Embedded Linux Packages

- ◆ KURT
 - > <http://www.ltc.ukans.edu/kurt>
- ◆ Linux Router Project
 - > <http://www.linuxrouter.org>
- ◆ Linux/RK
 - > <http://www.cs.cmu.edu/~raj कुमार/linux-rk.html>
- ◆ LOAF (Linux On A Floppy)
 - > <http://loaf.ecks.org>
- ◆ Linux-SRT
 - > <http://www.uk.research.att.com/~dmi/linux-srt/>
- ◆ Linux-VR
 - > <http://www.linux-vr.org>
- ◆ uCLinux
 - > <http://www.uclinux.org>
- ◆ uLinux (a.k.a muLinux)
 - > <http://sunsite.auc.dk/mulinux>
- ◆ PeeWeeLinux
 - > <http://www.peeweelinux.com>
- ◆ QLinux
 - > <http://www.cs.umass.edu/~lass/software/qlinux>
- ◆ RED-Linux
 - > <http://linux.ece.ucl.edu/RED-Linux>
- ◆ RTAI
 - > <http://www.rtai.org>

Embedded Linux Packages

- ◆ RTLinux
 - > <http://www.rtlinux.com>
 - ◆ ThinLinux
 - > <http://www.thinlinux.org>
 - ◆ Linux on Assabet
 - > <http://www.cs.cmu.edu/~wearable/software/assabet.html>
 - ◆ COMPAQ IPAQ Linux
 - > <http://handhelds.org>
 - ◆ Linux PPC Org
 - > <http://www.linuxppc.org>
 - ◆ Linux MIPS
 - > <http://linuxmips.ichilton.co.uk>
 - ◆ Linux SH
 - > <http://linuxsh.cjb.net>
 - > <http://www.m17n.org/linux-sh>
 - ◆ ARM Linux Project
 - > <http://www.arm.linux.org.uk/>
 - ◆ ELKS
 - > <http://www.elks.ecs.soton.ac.uk>
- From "A Survey of Embedded Linux Packages" by Rick Lehrbaum, Linux Journal, Feb. 2001*
more details...
<http://www.linuxdevices.com/articles/AT2760742655.html>

Packages (cont'd)

- CPU porting project
 - Linux-VR, Linux SH, Linux PPC org, ARM Linux Project
- Function enhancement project
 - Real-time Linux
 - KURT, Linux-SRT, QLinux, RED-Linux, ...
- Small-footprint Project
 - LOAF, ThinLinux, ELKS,...
- Open Source Product
 - Hard Hat Linux (Montavista)

Packages

- Proprietary product
 - Embedix (Lineo), BluecatLinux (LynuxWorks), ...
- IDE
 - Embedix (Lineo), BluecatLinux (LynuxWorks), ...
- Real-Time Scheduler Provider
- Micro Kernel Approach – Not True Linux
 - RTLinux, RTAI, ...

Linux Device Driver

Classes of Devices

- Character devices:
 - Character-based input/output
 - Accessed by means of filesystem nodes
 - Use open, close, read, write,,, system calls
 - The difference from a regular file
 - Regular file: can move back and forth
 - char devices: access sequentially
- Block devices
 - Like char devices, but accessed as multiples of a block (eg. disk).
- Network devices
 - Device exchanges data with other hosts

Linux File System Calls

```
int main()
{
    int fdi, fdo;
    char buf[100];
    ssize_t n;
    fdi = open("test.in", O_RDONLY);
    fdo = open("test.out", O_RDWR | O_CREAT);
    n = read(fdi, buf, 10);
    write(fdo, buf, n);
    lseek(fdi, 2, SEEK_CUR); // move the position of the file pointer by 2
    n = read(fdi, buf, 10);
    write(fdo, buf, n);
    close(fdi);
    close(fdo);
}
```

First Module Program

```
#define MODULE
#include <linux/module.h>
int init_module(void) { printk("Hello, world\n"); return 0; }
void cleanup_module(void) { printk("Goodbye, world\n"); }
▪ printk():
  ▪ defined in Linux kernel
  ▪ behaves similarly to standard C library function printf()
  ▪ Kernel does not use C library, thus needs its own print function
▪ Loading/unloading the module
  ➤ gcc -c hello.c // compile the module
  ➤ insmod ./hello.o // loading the module
  Hello, world // when the module loaded, execute init_module
  ➤ rmmod hello // unloading the module
  Goodbye, world // when the module lunoaded, execute
  // cleanup_module
```

printk

```
printk(KERN_DEBUG "Here I am %s:%i\n", __FILE__, __LINE__ & );  
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

- Similar to printf
- Difference: classify messages according to their severity by associating different loglevels
- Loglevels:
 - KERN_EMERG: emergency, preceding a crash
 - KERN_ALERT: requiring immediate action
 - KERN_CRIT: critical condition, related to HW or SW failures
 - KERN_ERR: error condition, related to HW difficulties
 - KERN_WARNING: problematic situation, may not create serious problems with the system
 - KERN_NOTICE: worthy of note. Security related conditions
 - KERN_INFO: informational message
 - KERN_DEBUG: used for debugging

Second Module Program

```
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/malloc.h>  
static int mrdrv_open(struct inode *inode, struct file *file)  
    { MOD_INC_COUNT; return 0; }  
static int mydrv_release(struct inode *inode, struct file *file)  
    { MOD_DEC_COUNT; return 0; }  
static ssize_t mydrv_read(struct file *file, char *buf, size_t  
    count,  
                          loff_t *ppos)  
    { printk("mydrv_read is invoked\n"); }
```

Second Module Program

```
static ssize_t mydrv_write(struct file *file, char *buf, size_t
    count, loff_t *ppos)
{ printk("mydrv_write is invoked\n"); }
struct file_operations mydrv_fops = {
    read: mydrv_read, write: mydrv_write, open: mydrv_open,
    release: mydrv_release }
int init_module(void) {
    int result;
    result = register_chrdev(250, "mydrv", &mydrv_fops);
    return result; }
void cleanup_module(void)
{ unregister_chrdev(250, "mydrv");
```

Second Module Program

- Compile
 - > gcc -D__KERNEL__ -DMODULE -c mydrv.c
- Load the module
 - > insmod mydrv.o
- Make the special device file
 - > mknod /dev/mydrv c 250 0

Second Module Program

```
▪ Test program
#include <fcntl.h>
#include MAX_BUFFER 26
char buf_in[MAX_BUFFER], buf_out[MAX_BUFFER];
int main() {
    int fd;
    fd=open("/dev/mydrv",O_RDWR);
    read(fd, buf_in, MAX_BUFFER);
    write(fd, buf_out, MAX_BUFFER);
    close(fd);
    return (0);
}
```

Scull

- Simple Character Utility for Loading Localities
- Character driver that acts on a memory area as a device
- Properties of the memory area:
 - Global: the data is shared by all the file descriptors that opened it.
 - Persistent: if the device is closed and reopened, data isn't lost.
- Scull0-scull3:
 - Four devices controlled by a single device driver
 - Basic type of device
- Scullpipe0-3, scullsingle, scullpriv, sculluid, scullwuid:
other variations of scull for the explanation of various Linux features

Device Structure of scull

```
typedef struct Scull_Dev {
    void **data;
    struct Scull_Dev *next; /* next listitem */
    int quantum;           /* the current quantum size */
    int qset;              /* the current array size */
    unsigned long size;
    devfs_handle_t handle; /* only used if devfs is there */
    unsigned int access_key; /* used by sculluid and scullpriv
    */
    struct semaphore sem; /* mutual exclusion semaphore
    */
} Scull_Dev;
```

Initialization in init_module()

- You need to register your device driver
- Example in the “mydrv” (the previous example)

```
int init_module(void) {
    int result;
    result = register_chrdev(MYDRV_MAJOR, “mydrv”,
    &mydrv_fops);
    return result; }
    ▪ MRDRV_MAJOR: the major number used by this driver
        ▪ each driver has its unique major number
    ▪ “mydrv”: the name of the driver
    ▪ &mydrv_fops: the function pointer used by this driverExample
    in scull initialization module
    result = register_chrdev(scull_major, "scull", &scull_fops);
```

Initialization in `init_module()`

- Function `register_chrdev()` updates the `MYDRV_MAJOR`'s entry of `chrdevs` table
 - Update `*name` and `*fops` field
- Character device switch table (in `fs/devices.c`)

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
}
.....
static struct device_struct chrdevs[MAX_CHRDEV];
.....
```
- Get information about the driver by from `/proc/devices`

File Operations Example

- Inside “mydrv”

```
struct file_operations mydrv_fops = {
    read: mydrv_read,
    write: mydrv_write,
    open: mydrv_open,
    release: mydrv_release
};
.....
int mrdrv_open(struct inode *inode, struct file *file) { ..... }
ssize_t mydrv_read(struct file *file, char *buf, size_t count,
    loff_t *ppos) { ..... }
.....
```

File Operations Example

- Inside scull

```
struct file_operations scull_fops = {
    llseek:  scull_llseek,
    read:    scull_read,
    write:   scull_write,
    ioctl:   scull_ioctl,
    open:    scull_open,
    release: scull_release,
};

.....
int scull_open(struct inode *inode, struct file *filp)
{ ..... }
ssize_t scull_read(struct file *filp, char *buf, size_t count,
    loff_t *f_pos) { ..... }
```

File Operations

```
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
    long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct inode *, struct dentry *, int);
```

File Operations

```
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, unsigned
    long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned
    long, loff_t *);
struct module *owner;
```

- `*owner`:
 - the only field that is not a function.
 - it's the pointer to the module that owns this structure
 - Set this field by
`SET_MODULE_OWNER (&scull_fops)` or
`owner: THIS_MODULE,`

Major and Minor Numbers

- Char devices are access through names in the filesystem
 - Called special files, device files, or nodes
 - Conventionally located in the `/dev` directory
eg `fd=open("/dev/mydrv", O_RDWR);`

- You need to creating a device node

```
mknod /dev/mydrv c 250 0
```

device name type major number minor number

- Major number identifies the driver
- Minor number is used only inside the driver
- This node is not removed until explicitly removing a device node

```
rm /dev/mydrv
```

Major and Minor Numbers

- You can see the devices by “ls -l”
- The output looks like

```
crw-rw-rw- 1 root root 1, 3 Feb 23 1999
null
crw----- 1 root root 10, 1 Feb 23 1999
psaux
.....
```
- Assignment of a major number
 - Assignment made at driver initialization (init_module) by calling `int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);`
 - major: major number being request
 - *name: name of the device to appear in /proc/devices
 - *fops: the pointer to an array of function pointers use to invoke your driver's entry points. Should be a global data structure within the driver, not to one local to init_module()

Dynamic Allocation

- Device major numbers already assigned to existing devices
- Dynamic assignment of a major number
 - When calling register_chrdev, the the argument 'major' to 0 → the function selects the free number and return it.
 - Register_chrdev returns
 - a positive number when major=0
 - 0 when major > 0
 - negative number for error
 - Disadvantage: can't create the device nodes in advance
 - After insmod, /proc/devices file includes the new device number
 - Check /proc/devices file to see the major number and then create the special files by calling

```
mknod /dev/scull0 c major_number 0
```
- For scull, a script scull_load is provided for dynamic allocation of major number

Dynamic Allocation

- Suggested way for major number assignment
 - Default: dynamic allocation
 - Option: specify the major number at load time or even at compile time

```
scull_major = SCULL_MAJOR // SCULL_MAJOR initialized
// in the head file

MODULE_PARM(scull_major,"i"); // make this variable available
// for module parameter → possible for modify at loadtime

.....
result = register_chrdev (scull_major, "scull", &scull_fops);
if (result < 0) {
    printk("scull: can't get major %d\n", scull_major);
    return result;
}
if (scull_major == 0) scull_major = result; // dynamic
```
- You can modify SCULL_MAJOR in the head file at compile-time

Parameter Assignment at Load

- Give a value to `scull_major` when calling `insmod()`

```
insmod scull scull_major=123
```
- Parameter values assigned at load time

```
insmod skull skull_ival=666 skull_sval="the beast"
```
- The module must make them available inside the program:

```
int skull_ival = 0;
char *skull_sval;

.....
MODULE_PARM (skull_ival, "i");
MODULE_PARM (skull_sval, "s");
```
- “i”: integer type, “s”: string, “b”: one byte, “h”: short (two bytes), “l”: long

Shutdown module

- Unregister any registered driver in `clean_module()`
- Inside “mydrv”

```
void cleanup_module(void)
{
    unregister_chrdev(MYDRV_MAJOR, “mydrv”);
}
```
- Inside “scull”

```
void scull_cleanup_module(void)
{
    int i;
    .....
    unregister_chrdev(scull_major, "scull");
    // deallocate any resource used by scull
}
```

Open Method

```
Scull_Dev *scull_devices;
.....
int scull_init_module(void) {
.....
scull_devices = kmalloc(scull_nr_devs * sizeof(Scull_Dev),
    GFP_KERNEL);
..... }
int scull_open(struct inode *inode, struct file *filp)
{
    Scull_Dev *dev;                // device information
    .....
    dev = &scull_devices[num];     // num: device minor number
    filp->private_data = dev;      // for other methods
    .....
}
```

Open Method

- Increment the usage count
- Check for device-specific errors
- Initialize the device if opened for the first time
- Identify the minor number and update the `f_op` pointer if necessary
- Allocate and fill any data structure to be put in `filp` → `private_data`

The Usage Count

- Keeps a usage count for every module to determine whether the module can be safely removed.
 - Module can't be unloaded if it is busy.
 - Three macros for the usage count
 - `MOD_INC_USE_COUNT` : Increments the count
 - `MOD_DEC_USE_COUNT` : Decrements the count
 - `MOD_IN_USE` : evaluates to true if the count is not zero
- ```
static int mrdrv_open(struct inode *inode, struct file *file)
 { MOD_INC_COUNT; return 0; }
static int mydrv_release(struct inode *inode, struct file *file)
 { MOD_DEC_COUNT; return 0; }
```
- `MOD_INC_USE_COUNT` called before doing almost anything else
  - The current value of the usage count is found in `/proc/modules`



# Open Method

```
int scull_open(struct inode *inode, struct file *filp)
{
 Scull_Dev *dev; /* device information */
 int num = NUM(inode->i_rdev);
 int type = TYPE(inode->i_rdev);
 /*
 * If private data is not valid, we are not using devfs
 * so use the type (from minor nr.) to select a new f_op
 */
 if (!filp->private_data && type) {
 if (type > SCULL_MAX_TYPE) return -ENODEV;
 filp->f_op = scull_fop_array[type];
 return filp->f_op->open(inode, filp); /* dispatch to specific open */
 }
}
```

# Open Method

```
/* type 0, check the device number (unless private_data
valid) */
dev = (Scull_Dev *)filp->private_data;
if (!dev) {
 if (num >= scull_nr_devs) return -ENODEV;
 dev = &scull_devices[num];
 filp->private_data = dev; /* for other methods */
}
MOD_INC_USE_COUNT; /* Before we maybe sleep */
/* now trim to 0 the length of the device if open was write-
only */
```

## Open Method

```
if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
 if (down_interruptible(&dev->sem)) {
 MOD_DEC_USE_COUNT;
 return -ERESTARTSYS;
 }
 scull_trim(dev); /* ignore errors */
 up(&dev->sem);
}
return 0; /* success */
}
```

## Release Method

- Deallocate anything that open allocated in `filp->private_data`
- Shut down the device on last close
- Decrement usage count

```
int scull_release(struct inode *inode, struct file *filp)
{
 MOD_DEC_USE_COUNT;
 return 0;
}
```

# Semaphore

- In scull, each device uses a semaphore

```
typedef struct Scull_Dev {

 struct semaphore sem; /* mutual exclusion semaphore */
} Scull_Dev;
```

- Initialize semaphore in `scull_init()`

```
for (i=0; i < scull_nr_devs; i++) {
 scull_devices[i].quantum = scull_quantum;
 scull_devices[i].qset = scull_qset;
 sema_init(&scull_devices[i].sem, 1);
}
```

- Obtain the semaphore

```
down_interruptible(&dev->sem)
```

- Release the semaphore

```
up(&dev->sem)
```

# User Space Data Movement

```
ssize_t read (struct file *filp, char *buff, size_t count, loff_t *offp);
```

```
ssize_t write (struct file *filp, char *buff, size_t count, loff_t *offp);
```

- Need to move data between the user space and the kernel space

```
unsigned long copy_to_user (void *to, const void *from, unsigned
 long count);
```

```
unsigned long copy_from_user (void *to, const void *from, unsigned
 long count);
```

- Need to update the file position

```
if (copy_to_user(buf, dptr->data[s_pos]+q_pos, count)) {
 ret = -EFAULT;
```

```
 goto out;
```

```
}
```

```
*f_pos += count;
```

- Returned value (when  $\geq 0$ ): the number of bytes successfully transferred

# The Read Method

```
ssize_t scull_read(struct file *filp, char *buf, size_t count,
 loff_t *f_pos)
{
 Scull_Dev *dev = filp->private_data; /* the first listitem */
 Scull_Dev *dptr;
 int quantum = dev->quantum;
 int qset = dev->qset;
 int itemsize = quantum * qset; /* how many bytes in the listitem */
 int item, s_pos, q_pos, rest;
 ssize_t ret = 0;
 if (down_interruptible(&dev->sem))
 return -ERESTARTSYS;
 if (*f_pos >= dev->size)
 goto out;
```

# The Read Method

```
 if (*f_pos + count > dev->size)
 count = dev->size - *f_pos;
 /* find listitem, qset index, and offset in the quantum */
 item = (long)*f_pos / itemsize;
 rest = (long)*f_pos % itemsize;
 s_pos = rest / quantum; q_pos = rest % quantum;

 /* follow the list up to the right position (defined elsewhere) */
 dptr = scull_follow(dev, item);

 if (!dptr->data)
 goto out; /* don't fill holes */
 if (!dptr->data[s_pos])
 goto out;
```

# The Read Method

```
/* read only up to the end of this quantum */
if (count > quantum - q_pos)
 count = quantum - q_pos;

if (copy_to_user(buf, dptr->data[s_pos]+q_pos, count)) {
 ret = -EFAULT;
 goto out;
}
*f_pos += count;
ret = count;

out:
up(&dev->sem);
return ret;
}
```

# The Write Method

```
ssize_t scull_write(struct file *filp, const char *buf, size_t count,
 loff_t *f_pos)
{
 Scull_Dev *dev = filp->private_data;
 Scull_Dev *dptr;
 int quantum = dev->quantum;
 int qset = dev->qset;
 int itemsize = quantum * qset;
 int item, s_pos, q_pos, rest;
 ssize_t ret = -ENOMEM; /* value used in "goto out" statements */

 if (down_interruptible(&dev->sem))
 return -ERESTARTSYS;
```

# The Write Method

```
/* find listitem, qset index and offset in the quantum */
item = (long)*f_pos / itemsize;
rest = (long)*f_pos % itemsize;
s_pos = rest / quantum; q_pos = rest % quantum;
/* follow the list up to the right position */
dptr = scull_follow(dev, item);
if (!dptr->data) {
 dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
 if (!dptr->data) goto out;
 memset(dptr->data, 0, qset * sizeof(char *));
}
if (!dptr->data[s_pos]) {
 dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
 if (!dptr->data[s_pos]) goto out;
}
}
```

# The Write Method

```
/* write only up to the end of this quantum */
if (count > quantum - q_pos)
 count = quantum - q_pos;
if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
 ret = -EFAULT;
 goto out;
}
*f_pos += count;
ret = count;
/* update the size */
if (dev->size < *f_pos)
 dev->size = *f_pos;
out:
up(&dev->sem);
return ret;
```

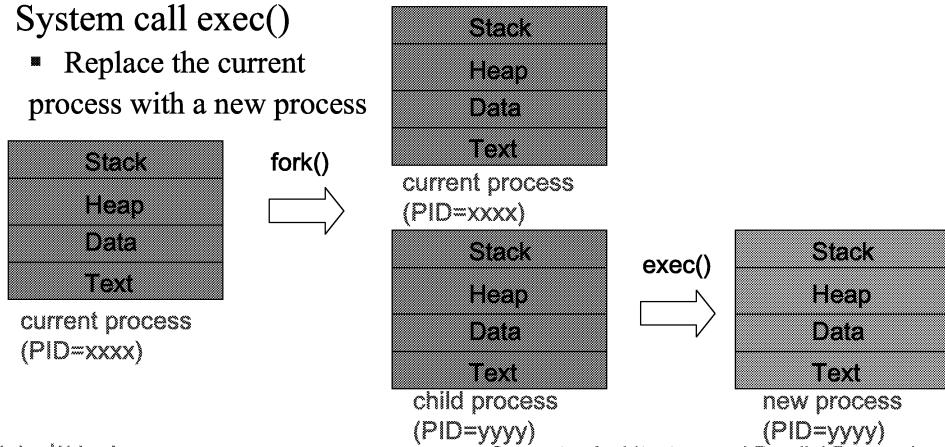
# Linux Features

# Task

- Basic building block of software
- States (used in this textbook)
  - Running: currently executed by the processor
  - Ready: not running but ready to run when processor is available
  - Blocked: cannot run even the processor is available
    - Eg) waiting for data to arrive
- Additional possible states: suspended, pended, waiting, dormant, delayed
- Scheduler:
  - Decides which task to run
  - Schedule the highest priority task first

# Creation of a Child Task

- System call `fork()`
  - Copy the current process to create a new process
  - The current process: parent process
  - The new process: child process
- System call `exec()`
  - Replace the current process with a new process



# Creation of a Child Task

- Example program

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(void) {
 printf("[%d] parent PID: %d\n", getpid(), getppid());
 fork();
 printf("[%d] parent PID: %d\n", getpid(), getppid());
}
```
- Execution result

```
[5964] parent ID: 5813
[5964] parent ID: 5813
[5965] parent ID: 5964
```



# Linux Task Identifiers

- Process (or task) identification number  
`#include <sys/types.h>`  
`pid_t getpid(void);`
- Process group identification number
  - Process group: a set of processes that perform a single job
    - May receive the same signal
    - Eg) `ps -ef | more`
      - “ps” and “more” are two processes in the same group`pid_t getpgrp(void);`
- Parent process identification number
  - Parent process: the process that creates the current process`pid_t getpid(void);`
- Parent group identification number:  
`pid_t getpgid (pid_t pid);`

# Linux Task Creation Example

- System call `fork()` returns
  - 0 in the child process and the child PID in the current process
- Example program:

```
.....
int main()
{
 pid_t pid;
 char *message;
 int n;
 printf("fork program starting\n");
 pid = fork ();
 switch (pid) {
 case -1:
 perror("fork failed");
 exit (1);
```

# Linux Task Creation Example

```
case 0:
 message = "This is the child";
 n = 5;
 break;
default:
 message = "This is the parent";
 n = 3;
 break;
}
for(; n>0; n--) {
 puts (message);
 sleep(1);
}
exit(0);
}
```

# Wait for Child Termination

- The parent process waits for the termination of its child process

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
 pid_t pid;
 char *message;
 int n;
 int exit_code;
 printf("fork program starting\n");
 pid = fork ();
```

# Wait for Child Termination

```
switch (pid) {
 case -1:
 perror("fork failed");
 exit (1);
 case 0: // child process
 message = "This is the child";
 n = 5;
 exit_code = 37;
 break;
 default: // parent process
 message = "This is the parent";
 n = 3;
 exit_code = 0;
 break;
}
```

# Wait for Child Termination

```
for(; n>0; n--) {
 puts (message);
 sleep(1);
}
if (pid != 0) { // parent process
 int stat_val; pid_t child_pid;
 child_pid = wait (&stat_val);
 printf("Child has finished: PID = %d\n", child_pid);
 if(WIFEXITED(stat_val))
 printf("Child exited with code %d\n",
 WEXITSTATUS(stat_val));
 else
 printf("Child terminated abnormally\n");
}
exit(exit_code);
```

# Simple Shell Program

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXLINE 255
#define NL 0x0A
int main (void) {
 char linebuf[MAXLINE];
 pid_t pid; int status;
 printf("$$ ");
 while (fgets (linebuf, MAXLINE, stdin) != NULL) {
 if ((strlen(linebuf) == 1) && (linebuf[0] == NL)) {
 printf ("$$ ");
 continue;
 }
 }
```

# Simple Shell Program

```
linebuf[strlen(linebuf)-1] = NULL;
if ((pid = fork()) < 0) {
 perror("fork");
 exit(1);
}
else if (pid == 0) { // child process
 execlp (linebuf, linebuf, (char *) 0);
 perror ("execlp");
 exit (2);
}
waitpid (pid, &status, 0); // wait for the process pid
printf ("$$ ");
}
exit(0);
}
```

# Signal

- Event that stimulates a process
- Registration of a signal handler

```
#include <signal.h>
```

```
void (*signal (int sig, void (*func)(int)))(int);
```

- sig: the signal to be processed, func: the signal handler

- Example

```
(void) signal (SIGINT, ouch); // jump to the function ouch()
// when SIGINT (terminal interrupt ^C) arrives
```

- Send a signal to another process

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill (pid_t pid, int sig);
```

- pid: the target process id the signal to be sent, sig: the signal type

- Example

```
kill (getppid(), SIGALRM);
```

# Signal

- Example program

```
#include <signal.h>
```

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
void ouch (int sig) {
```

```
 printf("OUCH! - I got signal %d\n", sig);
```

```
 (void) signal (SIGINT, SIG_DFL); // do default action
```

```
}
```

```
int main() {
```

```
 (void) signal (SIGINT, ouch); // jump to ouch() when
 // receives signal SIGINT
```

```
 while (1) {
```

```
 printf("Hello World!\n"); sleep(1);
```

```
 }
```

```
}
```

# Signal

- Execution result

```
./ctrlc
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! – I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

# Signal

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static int alarm_fired = 0;
void ding (int sig) {
 alarm_fired = 1;
}
int main() {
 pid_t pid;
 printf("alarm application starting\n");
 pid = fork();
```

# Signal

```
switch (pid) {
 case 0: // child process
 sleep(5);
 kill (getppid(), SIGALRM); // send a signal to the parent
 exit(0);
}
printf("waiting for alarm to go off\n");
(void) signal (SIGALRM, ding); // register signal handler

pause(); // wait till a signal arrives
if (alarm_fired) printf("Ding\n");

printf("done\n");
exit(0);
}
```

# Signal Interface

- Programming interface using signal

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *act,
struct sigaction *oact);
```

  - act: point to the signal handler
  - oact: stores the previous signal operations
- struct sigaction fields
  - void (\*) (int) sa\_handler: the signal handler
  - sigset\_t sa\_mask: the blocked signals
  - int sa\_flags: flags

# Signal

- Example program

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch (int sig) { printf("OUCH! - I got signal %d\n", sig); }
int main() {
 struct sigaction act;
 act.sa_handler = ouch; // set signal handler
 sigemptyset (&act.sa_mask); // clear the mask
 act.sa_flags = 0; // clear the flag
 sigaction (SIGINT, &act, 0);
 while (1) {
 printf("Hello World!\n"); sleep(1);
 }
}
```

# Pipes

- IPC (InterProcess Communication): data communication between processes
- The output of one process is linked to the input of the other
- System call

```
#include <unistd.h>
int pipe (int file_descriptor[2]);
 ▪ Data movement direction:
 file_descriptor[1] → file_descriptor[0]
 ▪ The other direction is not defined.
```



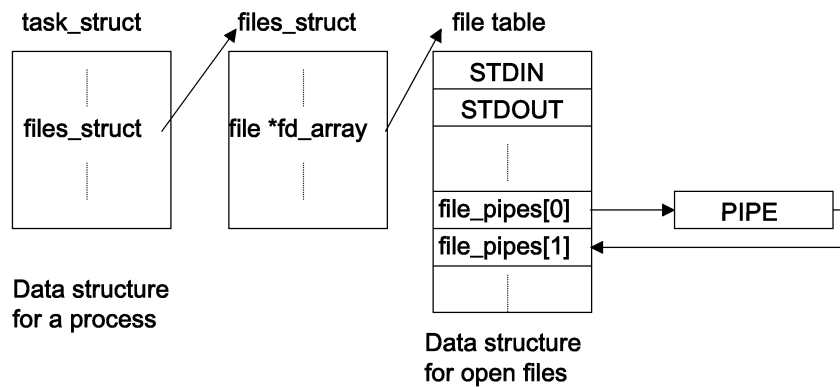
# Pipes

```
int main() {
 int data_processed;
 int file_pipes[2];
 const char some_data[] = "123";
 char buffer[BUFSIZ+1];
 memset(buffer, '\0', sizeof(buffer));
 if (pipe(file_pipes) == 0) {
 data_processed = write(file_pipes[1], some_data,
 strlen(some_data));

 printf("Wrote %d bytes\n", data_processed);
 data_processed = read(file_pipes[0], buffer, BUFSIZ);
 printf("Read %d bytes: %s\n", data_processed, buffer);
 exit(EXIT_SUCCESS);
 }
 exit(EXIT_FAILURE);
}
```

# Pipes

- Internal data structures



# Pipes

- Data communication between parent and child processes

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main() {
 int data_processed;
 int file_pipes[2];
 const char some_data[] = "123";
 char buffer[BUFSIZ+1];
 pid_t fork_result;

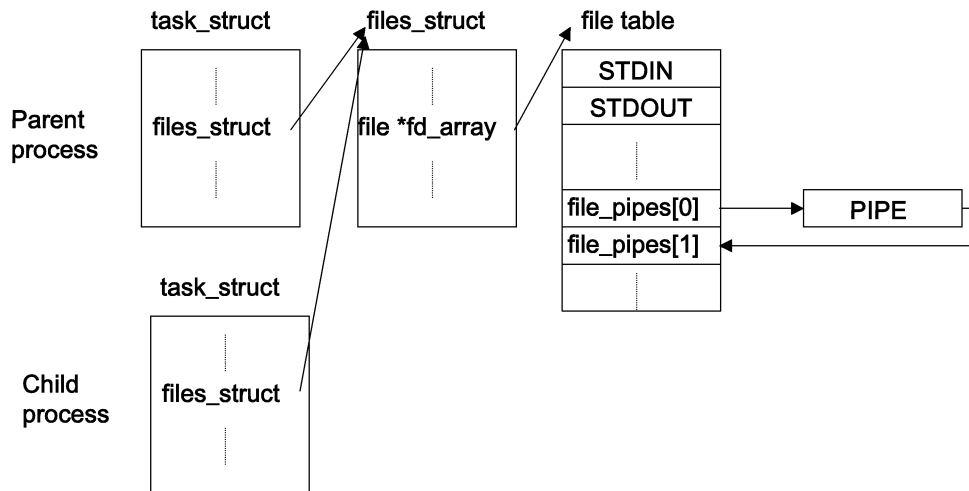
 memset(buffer, '\0', sizeof(buffer));
```

# Pipes

```
if (pipe(file_pipes) == 0) {
 fork_result = fork();
 if(fork_result == -1) {
 fprintf(stderr, "Fork failure");
 exit (EXIT_FAILURE); }
 if (fork_result == 0) { // child process
 data_processed = read(file_pipes[0], buffer, BUFSIZ);
 printf("Read %d bytes: %s\n", data_processed, buffer);
 exit(EXIT_SUCCESS);
 } else { // parent process
 data_processed = write(file_pipes[1], some_data,
 strlen(some_data));
 printf("Wrote %d bytes\n", data_processed);
 } }
exit(EXIT_SUCCESS);
```

# Pipes

- Internal data structures



# Pipes

- Data communication between different processes

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main() {
 int data_processed; int file_pipes[2];
 const char some_data[] = "123";
 char buffer[BUFSIZ+1]; pid_t fork_result;
 memset(buffer, '\0', sizeof(buffer));
 if (pipe(file_pipes) == 0) {
 fork_result = fork();
 if(fork_result == -1) {
 fprintf(stderr, "Fork failure");
 exit (EXIT_FAILURE); }
 }
```

# Pipes

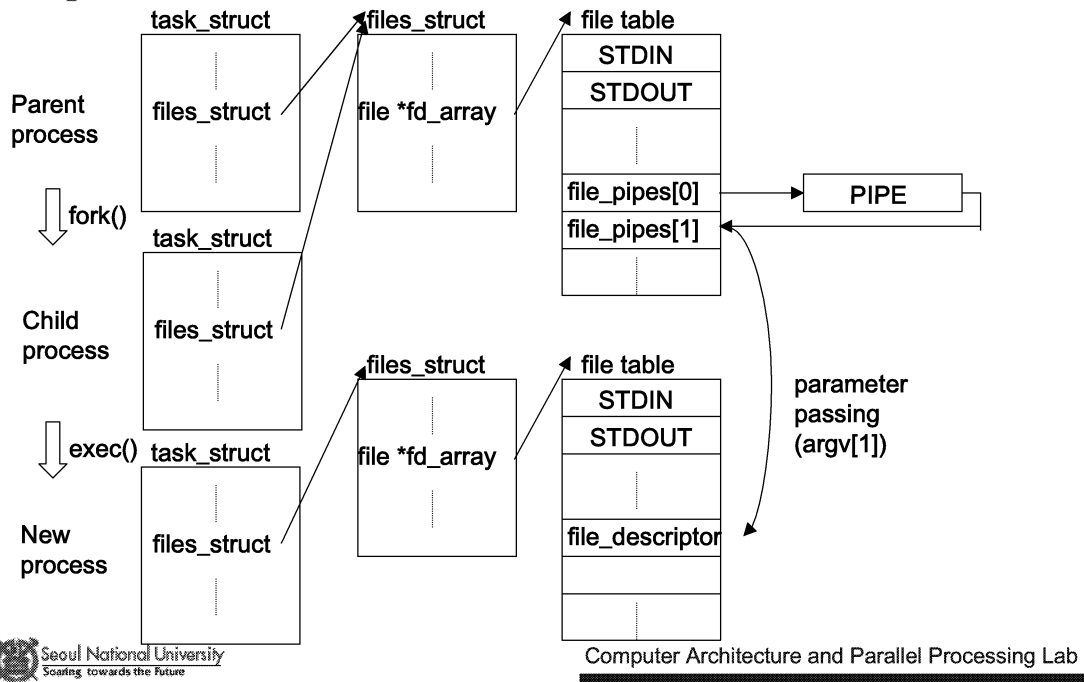
```
if (fork_result == 0) {
 sprintf(buffer, "%d", file_pipes[0]);
 (void)execl("pipe4", "pipe4", buffer, (char *)0);
 exit(EXIT_SUCCESS);
} else {
 data_processed = write(file_pipes[1], some_data,
 strlen(some_data));
 printf("Wrote %d bytes\n", data_processed);
}
}
exit(EXIT_SUCCESS);
}
```

# Pipes

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
 int data_processed;
 char buffer[BUFSIZ+1];
 int file_descriptor;

 memset(buffer, '\0', sizeof(buffer));
 sscanf(argv[1], "%d", &file_descriptor);
 data_processed = read(file_descriptor, buffer, BUFSIZ);
 printf("%d - read %d bytes: %s", getpid(), data_processed,
 buffer);
 exit(EXIT_SUCCESS);
}
```

# Pipes



# Pipes

```
#include <sys/types.h>
#include <sys/wait.h>
int main (argc, argv)
int argc; char *argv[];
{
 int fildes[2], status;
 if (pipe(fildes) == -1) {
 perror("pipe"); exit(1);
 }
 if (fork() == 0) { /* first child */
 close(fildes[0]);
 dup2(fildes[1], STDOUT_FILENO);
 close(fildes[1]);
 execlp("who", "who", (char *)0);
 exit(2);
 }
}
```

# Pipes

```
if (fork() == 0) { /* second child */
 close(fildes[1]);
 dup2(fildes[0], STDIN_FILENO);
 close(fildes[0]);
 execlp("sort", "sort", (char *)0);
 exit(3);
}

close (fildes[0]);
close (fildes[1]);

while (waitpid(0, &status,0) != -1);
exit(0);

}
```

# Named PIPE: FIFO

- Create a pipe  
\$ mkfifo filename

- Or fifo1.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
int main()
{
 int res = mkfifo ("/tmp/my_fifo", 0777);
 if (res == 0) printf("FIFO created\n");
 exit (EXIT_SUCCESS);
}
```

# Named PIPE: FIFO

- You can check the behavior of the pipe with general file access commands

```
$ ls -l /tmp/my_fifo
```

```
prwxrwxr-x 1 hjee 0 Aug 23 18:37
/tmp/my_fifo
```

```
$ cat < /tmp/my_fifo &
```

```
$ echo "sdsdfasdf" > /tmp/my_fifo
```

```
sdsdfasdf
```

```
[1]+ Done cat < /tmp/my_fifo
```

```
$
```

# Named PIPE: FIFO

- `open(const char *path, O_RDONLY)`
  - Open blocked until the FIFO is opened for WRITE by a certain process
  - Read blocked until data is available
- `open(const char *path, O_RDONLY | O_NONBLOCK)`
  - Open succeeds immediately whether the FIFO is opened for write or not
  - Read returns 0 immediately when no data is available
- `open(const char *path, O_WRONLY)`
  - Open blocked until the FIFO is opened for READ by a process
  - Write blocked until data can be written
    - If FIFO cannot store all data,
      - Return failure if the number of requested data is less than PIPE\_BUF
      - Return a part of data if the number of requested data is greater than PIPE\_BUF

# Named PIPE: FIFO

- `open(const char *path, O_WRONLY | O_NONBLOCK)`
  - Return immediately
  - Succeed if the FIFO is opened for read
  - Return `-1` otherwise
- For `O_RDONLY` and `O_WRONLY`, two processes are synchronized at the open call.

# FIFO

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
int main (int argc, char *argv[])
{
 int res;
 int open_mode = 0;
 if (argc < 2) {
 fprintf(stderr, "Usage: %s <some combination of \
O_RDONLY O_WRONLY O_NONBLOCK.\n", *argv);
 exit (EXIT_FAILURE);
 }
}
```



# FIFO

```
argv++;
if (strncmp (*argv, "O_RDONLY", 8) == 0) open_mode |= O_RDONLY;
if (strncmp (*argv, "O_WRONLY", 8) == 0) open_mode |= O_WRONLY;
if (strncmp (*argv, "O_NONBLOCK", 10) == 0) open_mode |=
O_NONBLOCK;
argv++;
if (*argv) {
 if (strncmp (*argv, "O_RDONLY", 8) == 0) open_mode |=
O_RDONLY;
 if (strncmp (*argv, "O_WRONLY", 8) == 0) open_mode |=
O_WRONLY;
 if (strncmp (*argv, "O_NONBLOCK", 10) == 0) open_mode |=
O_NONBLOCK;
}
```

# FIFO

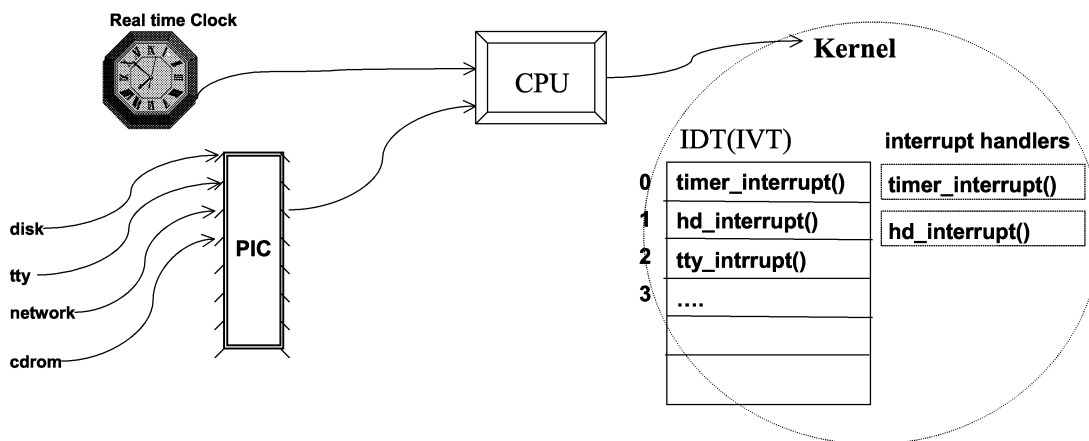
```
if (access(FIFO_NAME, F_OK) == -1) {
 res = mkfifo (FIFO_NAME, 0777);
 if (res != 0) {
 fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
 exit (EXIT_FAILURE);
 }
}

printf("Process %d opening FIFO\n", getpid());
res = open (FIFO_NAME, open_mode);
printf("Process %d result %d\n", getpid(), res);
sleep (5);
if (res != -1) (void) close (res);
printf("Processo %d finished\n", getpid());
exit (EXIT_SUCCESS);
}
```

# Linux Internal

# Interrupt in Linux

- Mechanism that peripherals signal an asynchronous event to the Kernel



# Interrupt Handling

- A peripheral generates an interrupt to the PIC (Programmable Interrupt Controller)
- PIC generates an interrupt to the CPU
- CPU requests the PIC the interrupt number (the interrupting device)
- PIC provides the interrupt number with the CPU
- The kernel inspects the IDT (Interrupt Descriptor Table) with the interrupt number as the index
- The kernel jumps to the interrupt service routine
- Save the program context
- Process the interrupt
- Restore the program context

# Trap

- Software version of an interrupt

| idt table |                   |
|-----------|-------------------|
| 0         | divide_error()    |
| 1         | debug()           |
| 2         | nmi()             |
|           | ....              |
|           |                   |
| 20        | timer_interrupt() |
| 21        | hd_interrupt()    |
|           | ....              |
|           |                   |
|           |                   |
| 80        | system_call()     |
|           | ....              |

Common trap handler for 8086

Device interrupt handler (IRQ)

# New System Call

- Modify “include/asm-i386/unistd.h” file

```
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
.....
#define __NR_vfork 190
#define __NR_newsyscall 191
```
- Modify sys\_call\_table (arch/i386/kernel/entry.S)

```
#define __NR_exit 1
.long SYMBOL_NAME (sys_ni_syscall) /* 0 */
.long SYMBOL_NAME (sys_exit) /* 1 */
.long SYMBOL_NAME (sys_fork) /* 2 */
.....
.long SYMBOL_NAME (sys_vfork) /* 190 */
.long SYMBOL_NAME (sys_newsyscall) /* 191 */
.rept NR_syscall-191
```

# New System Call

- New system call:

```
/* /usr/src/linux/kernel/newfile.c */
#include <linux/unistd.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/sched.h>
asmlinkage int sys_newsyscall()
{
 printk (“Hello Linux, I’m in Kernel\n”);
 return (0);
}
```

# New System Call

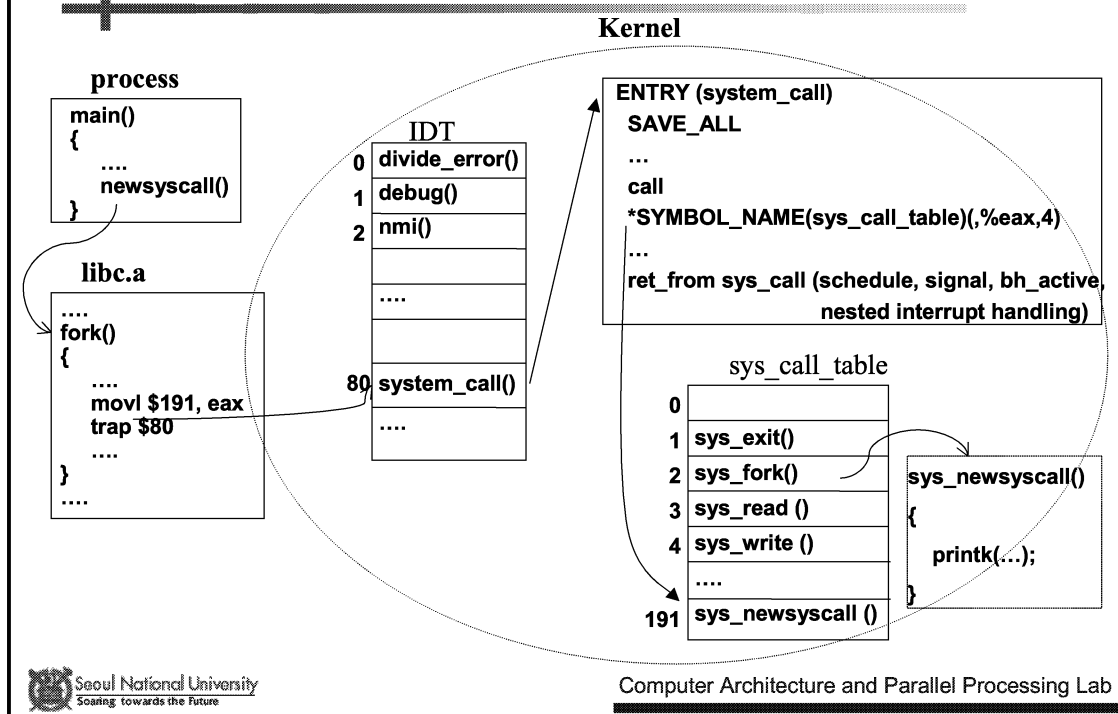
- Test program

```
#include <linux/unistd.h>
_syscall0 (int, newsyscall);
```

```
main()
{
 int i;
 i = newsyscall();
}
```

```
#define _syscall0 (type, name) \
type name (void) { \
 long __res \
 __asm__ volatile ("int 0x80 \
 : "=a" (__res) \
 : "0" (__NR ##name)); \
 __syscall_return (type, __name); \
}
```

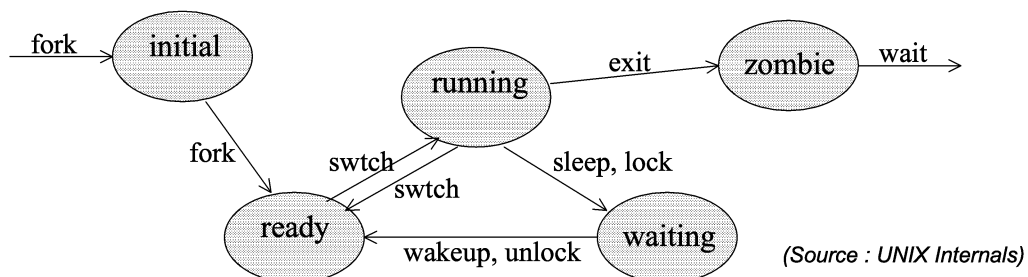
# New System Call



# Task Management

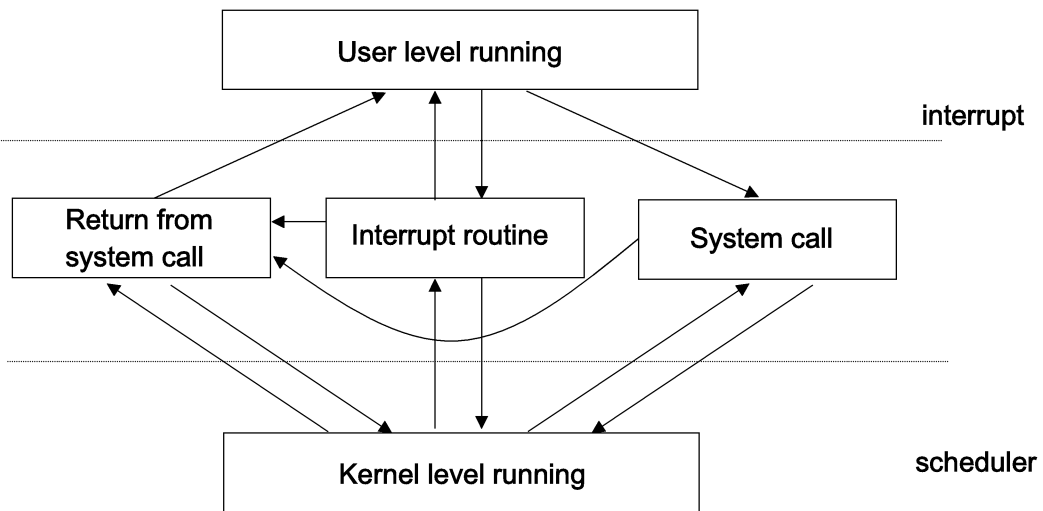
- Task:
  - Running or runnable program
  - May be used the same meaning as “process”
- Data structure task\_struct: the kernel internal data structure allocated to each task
- System call fork(): create a new task
  - Create a new task\_struct data structure and initialize it (mainly copy from its parent task)
  - All tasks except “pid 0” is created by fork()
- System call execve(): replace the current task
  - In general, when you run a new program, you call fork() and then call execve()
- Task removal
  - System call exit()
  - Killed by other process (eg. signal generated by kill())
  - Abnormal state (eg. segmentation fault)

# Task State and Transition

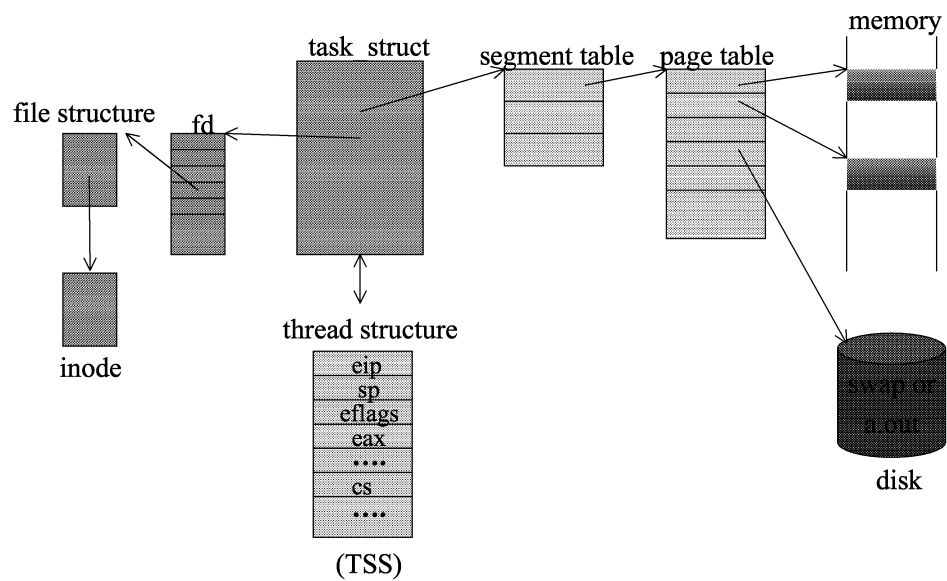


- Zombie state:
  - Return most of resources to the kernel
  - Save information about the process
  - The parent process will kill the process later

# Task State and Transition



# Task Context



# Data Structure task\_struct

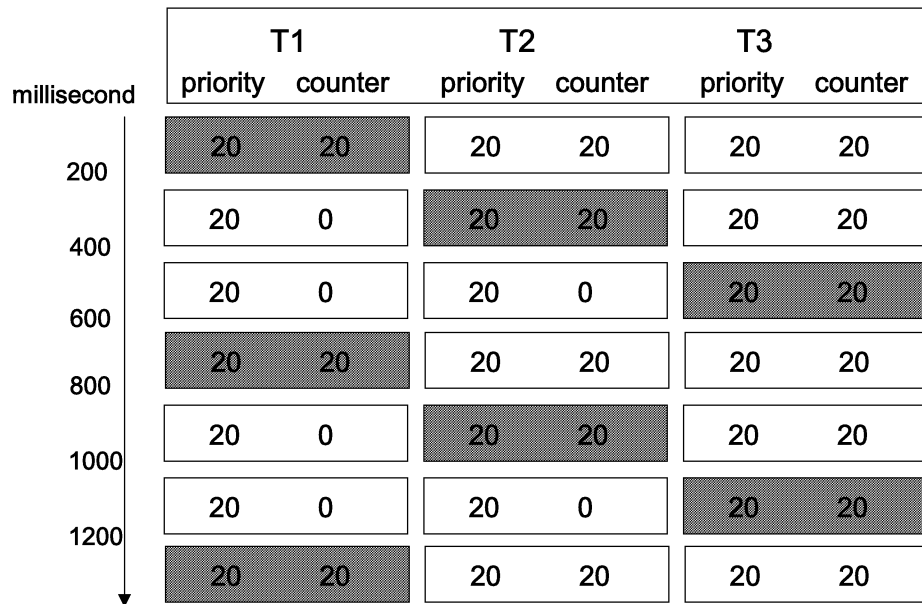
- /include/linux/sched.h
- Task identification: (eg: pid, uid )
- Task state: state
- Task relationship
  - Eg: p\_pptr (pointing the parent task)
  - Eg: all tasks are doubly-linked list: next\_task, prev\_task
  - Eg: all running or runnable tasks (TASK\_RUNNING) are double-linked list: next\_run, prev\_run
- Scheduling information: (eg: policy, priority, counter, need\_resched)
- Signal related information (eg: signal\_struct, sigpending, signal, blocked)
- Memory information: mm\_struct (text, data, stack, heap location and size, access control, page table information)
- File information: files\_struct (files opened by the task)
- Thread structure: thread\_struct tss (store the context: program counter, stack pointer, general register contents, flags, page directory, etc)
- Miscellaneous

# Scheduling

- Scheduling class: policy
  - SCHED\_FIFO: real-time nonpreemptive scheduling
  - SCHED\_RR: real-time preemptive scheduling
  - SCHED\_OTHER: non real-time time-sharing
- Schedule(): in kernel/sched.c
  - called when
    - The current task is moved to “wait” state
    - need\_resched is set to ‘1’
  - Schedule real-time task first with the highest rt\_priority
  - Schedule non real-time task with the highest priority + counter
    - The current processor p\_counter: decremented by timer tick
  - If (counter == 0) for all tasks,
    - counter = priority for all tasks
  - Context switch: switch\_to (current, next) /\*  
arch/i386/kernel/process.c \*/

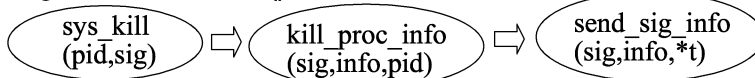


# Scheduling

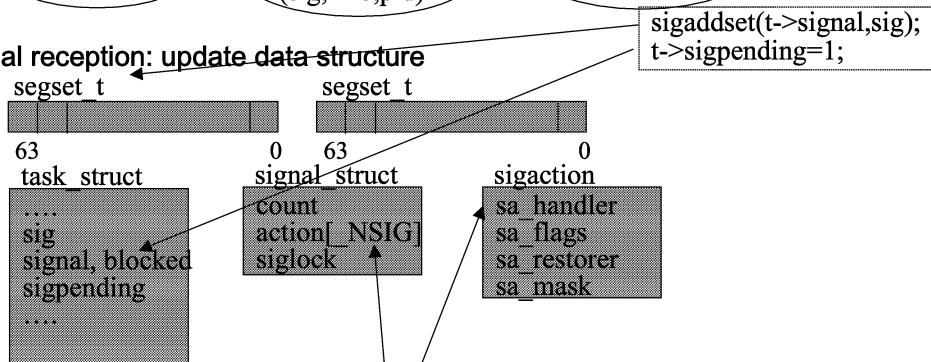


# Signal

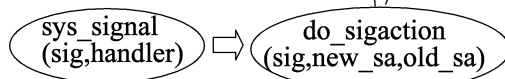
Signal generation: call kill()



Signal reception: update data-structure



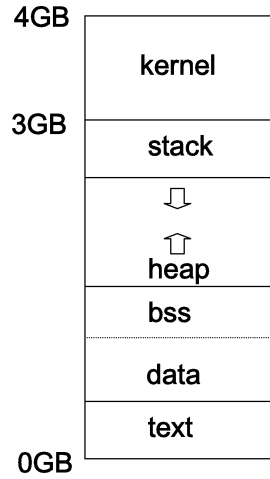
Signal processing: call signal()



# Memory Management

- Virtual memory:
  - 4GB memory space is given to a user regardless of the physical memory size
- Segments
  - Text: instructions
  - Data: global variables
    - BSS: uninitialized data
  - Stack: local variables
  - Heap: dynamic memory allocation
- Page:
  - fixed size (4KB)
  - A segment consists of one or more pages

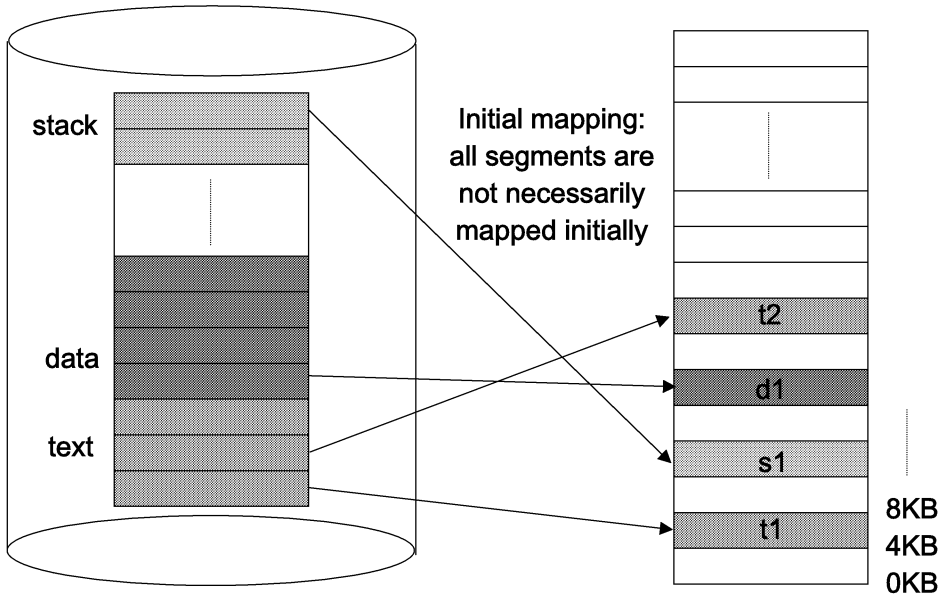
Program virtual memory space



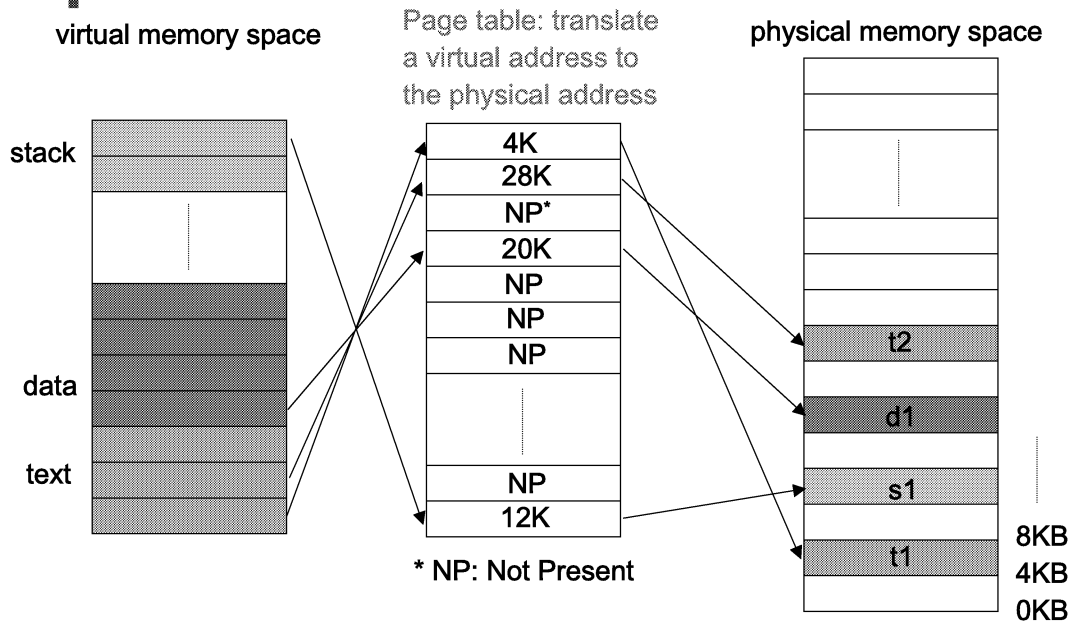
# Memory Management

virtual memory space

physical memory space



# Page Table

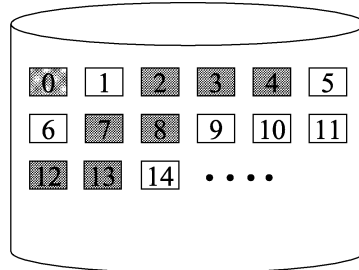


# Data Structure: mm\_struct

- A member of task\_struct
- Segment management: vm\_area\_struct
  - Represents each segment
  - All segments are double linked listed
  - vm\_start (vm\_end): the start (end) address of the segment
  - vm\_file: the file in which the segment is stored
  - vm\_offset: the location of the segment in the file indicated by vm\_file
- Page directory start address: pgd
- Virtual memory variables:
  - start\_code (end\_code): the start (end) address of text segment
  - start\_data (end\_data): the start (end) address of data segment
  - start\_brk (brk): the start (end) address of heap
  - start\_env (end\_env): the start (end) address of environments
  - start\_arg (end\_arg): the start (end) address of arguments
  - start\_stack: the start address of stack

# File System

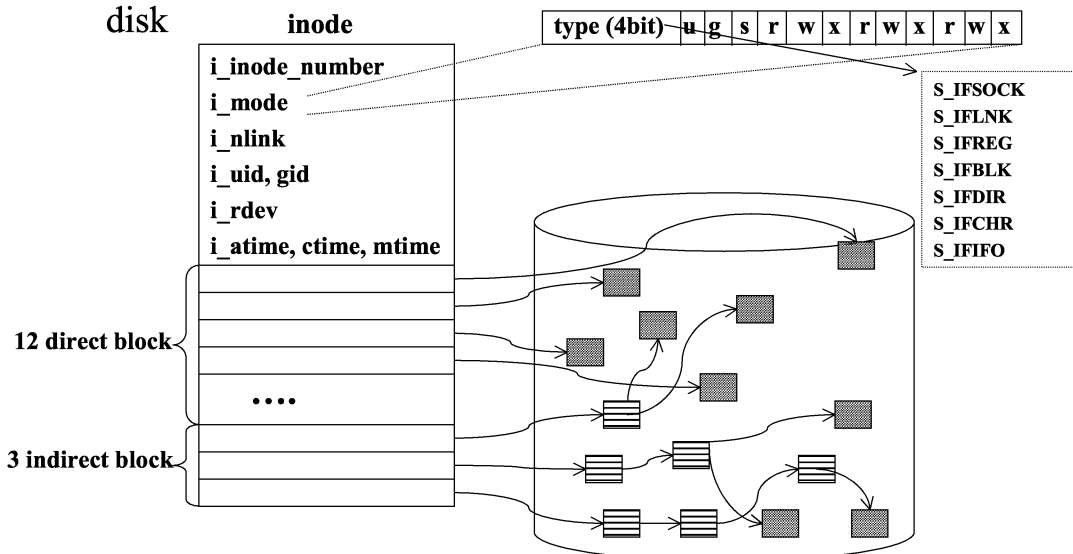
- File system:
  - a collection of logical disk blocks
  - the disk block size = the page frame size (4KB for intel processor)



- Disk block allocation
  - Sequential allocation: allocate to consecutive blocks
  - Nonsequential allocation
    - Block chain:
    - Indexed block
    - FAT (File Allocation Table)

# Data Structure: inode

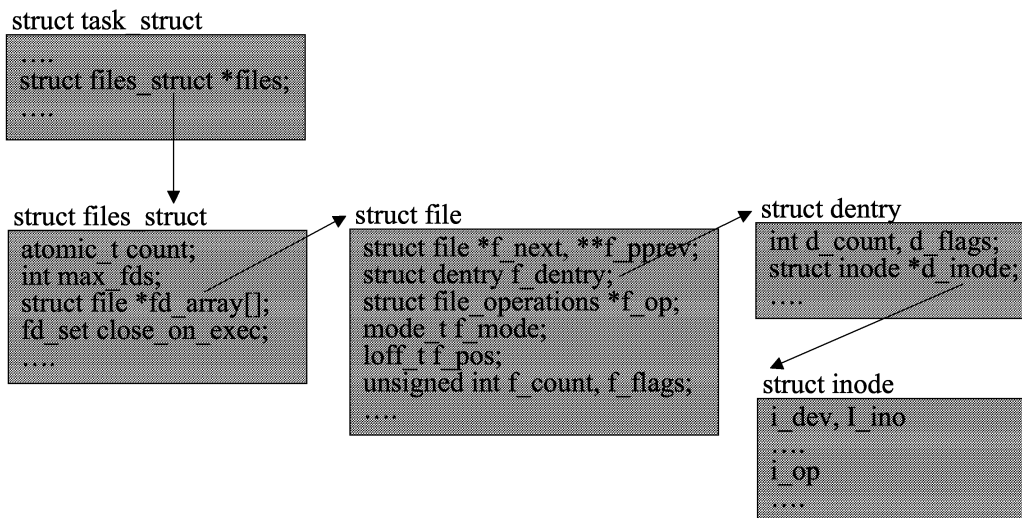
- The kernel internal data structure allocated to each file in disk



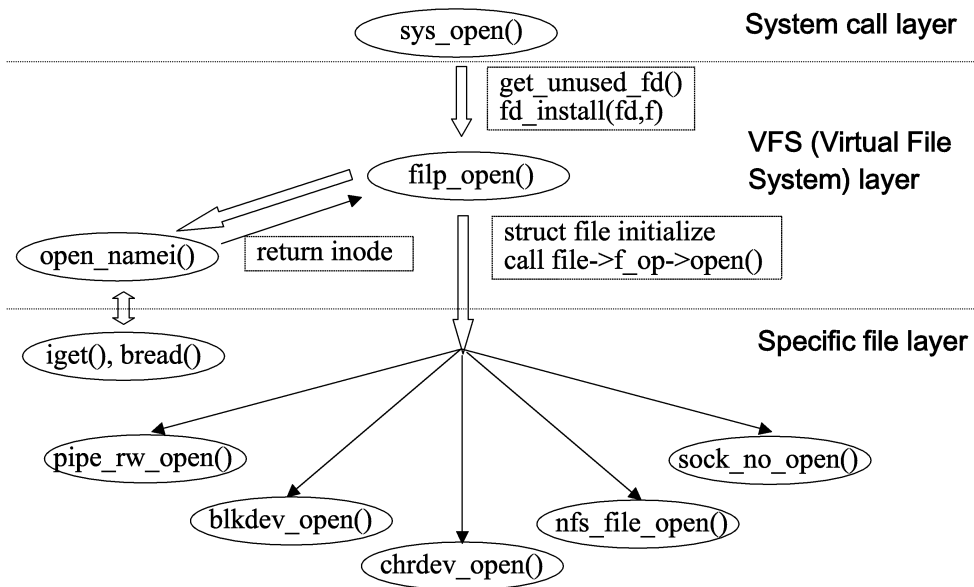
# Data Structure: inode

- The size of file supported by inode structure
  - Assume page size: 4-KB and pointer size 4-byte
  - 12 direct blocks:  $4\text{KB} \times 12 = 48\text{KB}$
  - 3 indirect blocks
    - 1<sup>st</sup> block:
      - $4\text{KB size}/4\text{-byte-pointer} = 1\text{K pointers}$
      - $1\text{K} \times 4\text{KB} = 4\text{MB}$
    - 2<sup>nd</sup> block:  $1\text{K} \times 1\text{K} \times 4\text{KB} = 4\text{GB}$
    - 3<sup>rd</sup> block:  $1\text{K} \times 1\text{K} \times 1\text{K} \times 4\text{KB} = 4\text{TB}$
  - In total:  $48\text{KB} + 4\text{MB} + 4\text{GB} + 4\text{TB}$
- The file size supported by Linux: 4 GB

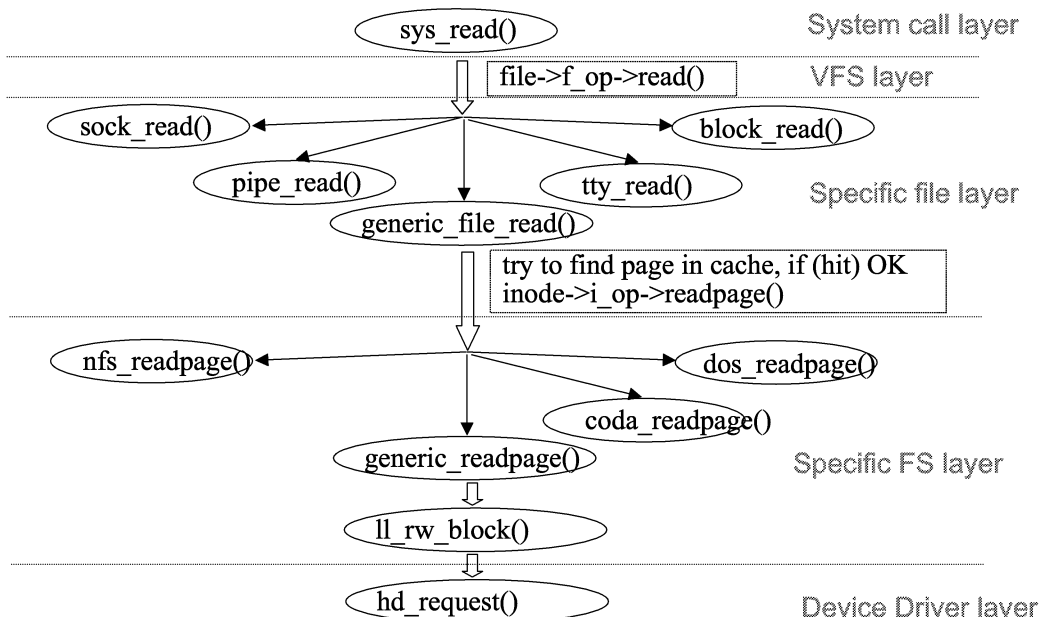
# User Interface



# File System Call Flow



# File System Call Flow



# Real-Time Scheduling

## Clock-driven Approach

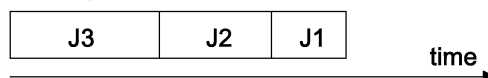
- Also called time-driven
- Scheduling decision is made at specific time instants
  - These instants chosen a priori before the system begins execution
  - Typical system
    - Schedule is computed off-line and stored for use at run time
    - The system is deterministic with a few exception
    - Scheduling overhead is minimized
  - Another choice
    - Make scheduling decision periodically using timer.

# Weighted Round-robin

- Round-robin approach
  - Used for time-shared applications
  - Jobs inserted in a FIFO queue when ready
  - The job at the head executes for one time slice
  - If job does not complete, it is preempted and placed at the end of the queue
  - Also called processor-sharing algorithm
- Weighted round-robin approach
  - Different jobs may have different weights (i.e., assigned job slots)
  - Control the speed of each job by assigning a weight of a job

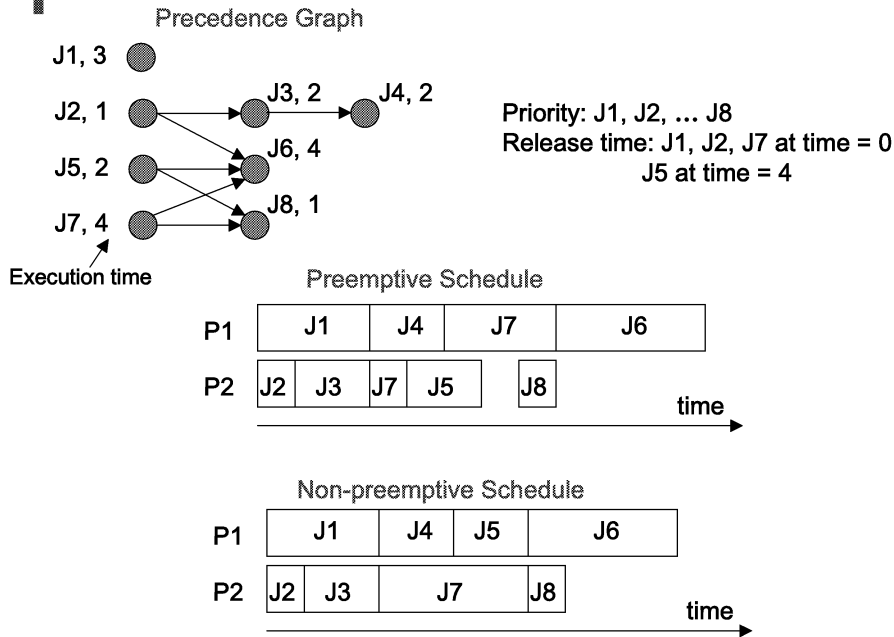
# Priority-driven Approach

- Scheduling decision made at the occurrences of events such as releases and completions of jobs
- Jobs are prioritized and the highest priority job is scheduled at any decision time.
- Example)
  - FIFO, LIFO: prioritized based on the release time
  - SETF (Shortest-Execution-Time-First), LETF (Longest-Execution-Time-First): prioritized based on the execution time
  - EDF (Earliest Deadline First): prioritized based on its dead line
    - Example)
      - Job#1 (J1): execution time=2, deadline=10
      - Job#2 (J2): execution time=3, deadline=7
      - Job#3 (J3): execution time 4, deadline=5





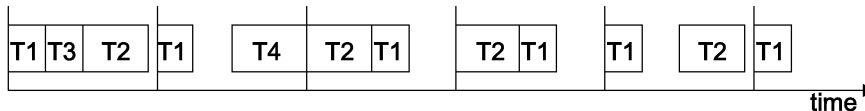
# Priority-driven Example



# Scheduling of Periodic Tasks

## Example:

- Task#1:  $T_1=(4,1)$  (period=4, execution time=1)
- $T_2=(5,1.8)$ ,  $T_3=(20,1)$ ,  $T_4=(20,2)$
- Hyperperiod (least common multiples of all periods) = 20
- An arbitrary schedule



- Unused intervals (eg. (3.8,4), (5,6)):
  - Scheduled for aperiodic jobs
  - Better to spread out these unused intervals
- A clock-driven schedule
  - $(t_k, T(t_k))$ :  $t_k$  – a decision time,  $T(t_k)$ -the task to be executed at  $t_k$
  - Eg) (1,  $T_3$ ), (2,  $T_2$ ), (3.8, Idle), (4,  $T_1$ ), ..., (19.8, Idle)
  - The number of events = 17, The hyperperiod = 20

# A Cycle Scheduler

Input: Stored schedule  $(t_k, T(t_k))$  for  $k=0,1,\dots,N-1$  ( $N$ :# of events in a hyperperiod  $H$ )

Task SCHEDULER

```

set the next decision point i and table entry k to 0
set the timer to expire at t_k
do forever
 accept timer interrupt;
 if an aperiodic job is executing, preempt the job;
 current task $T=T(t_k)$;
 increment i by 1;
 compute the next table entry $k=i \bmod (N)$;
 set the timer to expire at $\lfloor i/N \rfloor H + t_k$;
 if the current task T is I (=idle),
 let the job at the head of the aperiodic queue execute;
 else let the task T execute
 sleep;

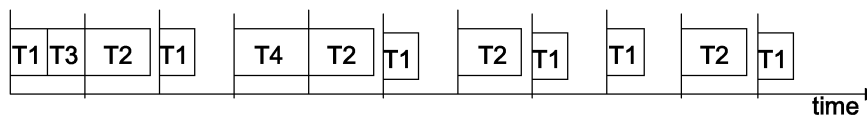
```

end SCHEDULER

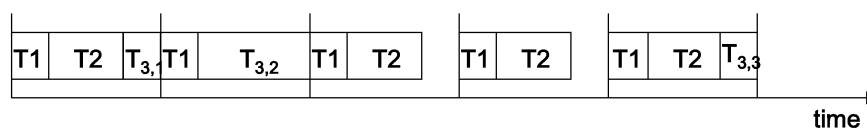


# Cyclic Executive Schedule

- Scheduling decisions are made not arbitrary time but at the beginning of every frame
  - Example) frame size = 2

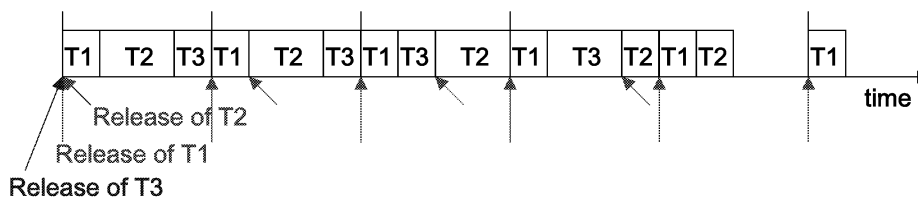


- Decompose job into slices
  - Example)  $T1=(4,1)$ ,  $T2=(5,2)$ ,  $T3=(20,5)$
  - Decompose  $T3$  into  $T3,1=(20,1)$ ,  $T3,2=(20,3)$ ,  $T3,3=(20,1)$
  - Frame size = 4



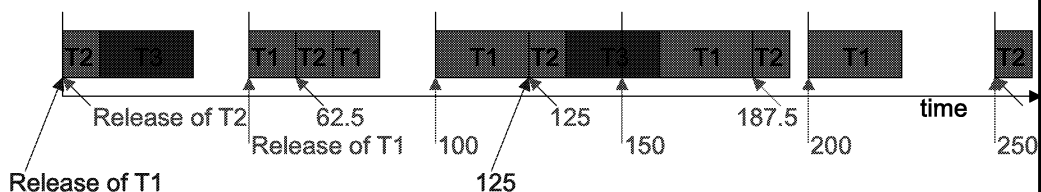
# Rate Monotonic (RM)

- Static Priority Assignment
- Assigned priority based on its rate (the inverse of the period)
- The higher the rate, the higher the priority
- Example)  $T1=(4,1)$ ,  $T2=(5,2)$ ,  $T3=(20,5)$ 
  - Priority:  $T1 \rightarrow T2 \rightarrow T3$
  - Each job is placed at the priority queue as soon as released
  - Each job is executed as soon as it is at the head of the priority queue



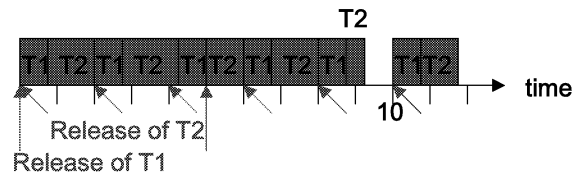
# Deadline Monotonic (DM)

- Static Priority Assignment
- Assigned priority based on its relative deadline (=deadline/period)
- The shorter the relative deadline, the higher the priority
- Example)
  - $T1=(50,50,25,100)$  =(phase, period, execution time, relative deadline),  $T2=(0,62.5,10,20)$ ,  $T3=(0,125,25,50)$
  - Priority:  $T2 \rightarrow T3 \rightarrow T1$
  - Note: You cannot meet the deadline with the RM schedule



# Dynamic Priority Assignment

- Earliest Deadline First (EDF)
  - The closer absolute deadline, the higher priority
  - Example)  $T1=(2,0.9)$ ,  $T2=(5,2.3)$



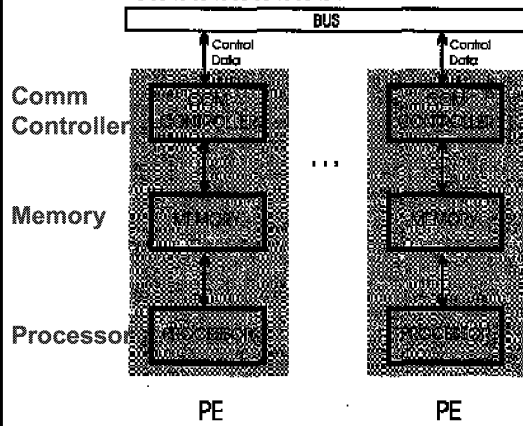
- Least Slack Time First (LST)
  - The remaining execution time:  $x$
  - Deadline:  $d$ , Current time:  $t$
  - Slack =  $d - t - x$
  - The smaller slack, the higher priority
  - Example) Exactly the same schedule as above

# Scheduling and Bus

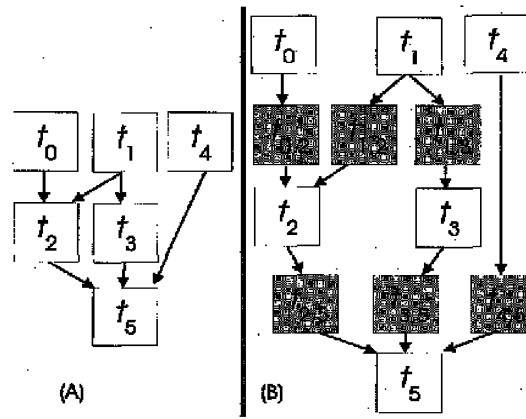
- SW job access HW component through a bus
- Bus contention increases SW job processing time
  - No guarantee of a bus access within a deadline
- Bus arbitration scheme for given target application necessary
  - Control area network: priority = deadline
  - Multimedia stream processor: Bus arbitration-SW co-scheduling

# Scheduling and Bus

- Target architecture dependence



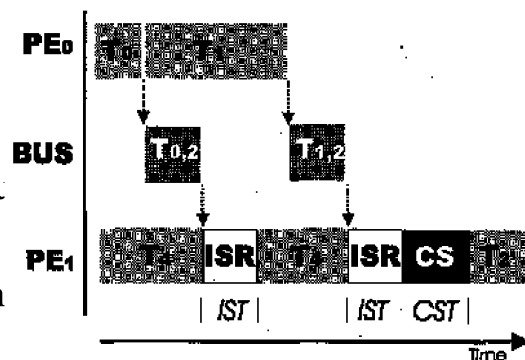
Task and



(communication arc  $\rightarrow$  communication task)

# Execution Model

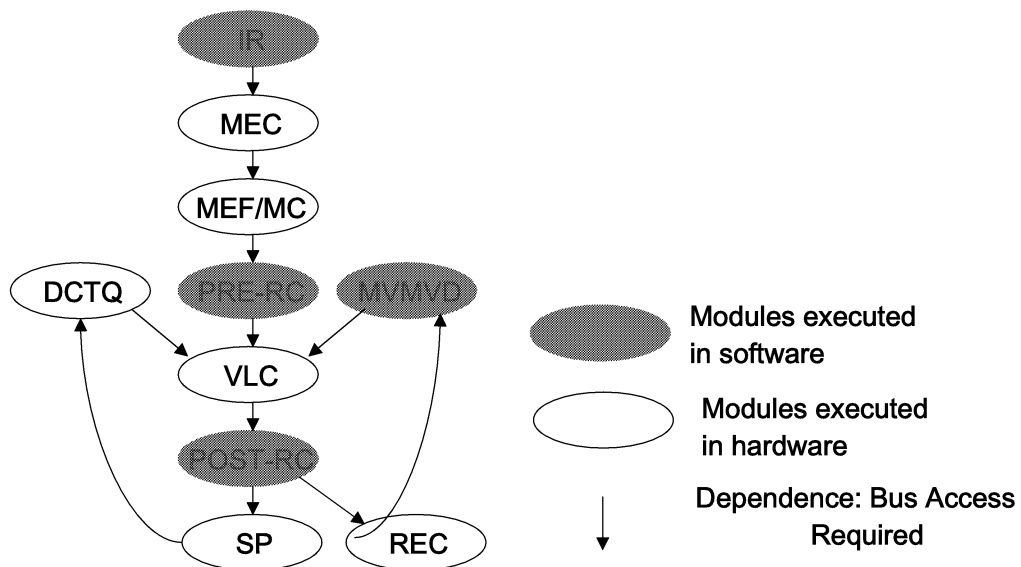
- Assumption
  - PE0: T0, T1
  - PE1: T4, T2
  - Priority: T0 > T1, T2 > T4
- ISR: Interrupt Service Routine
- CS: context switching
- $T_{0,2}$ : communication from PE0 to PE2
- Simultaneous execution of T1 and T0,2 (communication) requires dual port RAM



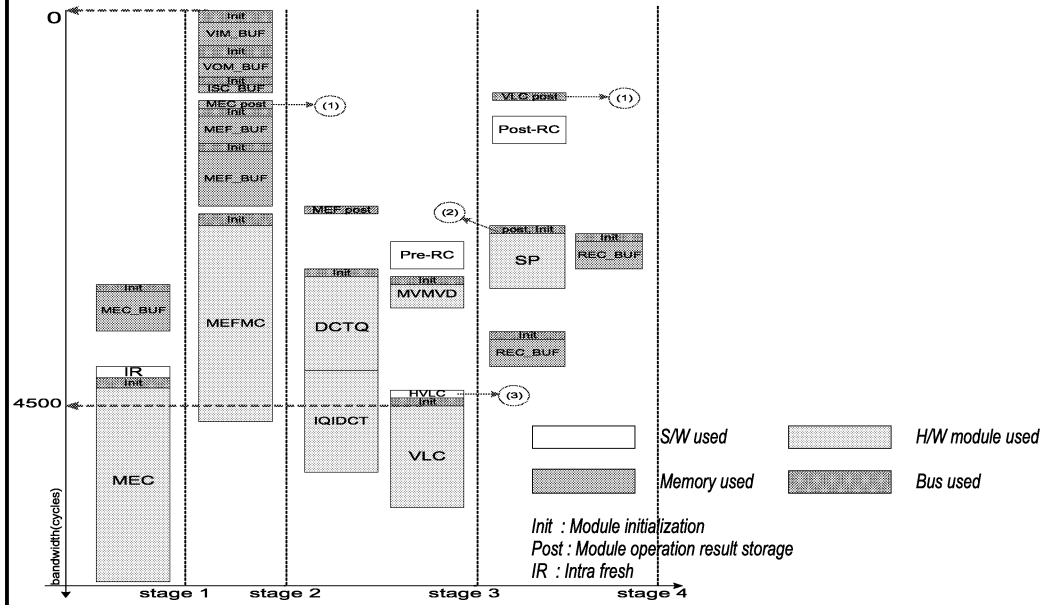
# Scheduling Procedure

- Initial scheduling: random task allocation to PE and priority assignment
- Iterative improvement
  1. Utilization update: PE or bus is over-utilized → task reallocation
  2. Speed update: a task with deadline violation → rescheduled/allocated
- D.L. Rhodes, W. Wolf, "Overhead Effects in Real-time Preemptive Schedules," in CODES99, pp. 193-197.

# MPEG-4 Encoding Job



# Scheduling and Arbitration



# Linux Device Driver Programming

## Blocking I/O

- Question: What to do when there's no data yet
- Answer: go to sleep waiting for data
- Sleeping:
  - suspend execution
  - freeing the processor for other uses
  - at some time later, wake up
  - continue the job when the event being waited for occurs.
- Wait queue: a queue of processes waiting for an event.  
`wait_queue_head_t my_queue; // declaration`  
`init_waitqueue_head (&my_queue); // initialization`
- Another declaration method for a static variable  
`DECLARE_WAIT_QUEUE_HEAD (my_queue);`



# Blocking I/O

- Various sleeps depending on how deep a sleep is called for:  
sleep\_on (wait\_queue\_head\_t \*queue);
  - Put the process to sleep on this queue. Not interruptibleinterruptible\_sleep\_on (wait\_queue\_head\_t \*queue);
  - Interruptible version of sleep\_onsleep\_on\_timeout (wait\_queue\_head\_t \*queue, long timeout);  
interruptible\_sleep\_on\_timeout (wait\_queue\_head\_t \*queue, long timeout);
  - Sleep will last no longer than the given timeoutvoid wait\_event (wait\_queue\_head\_t queue, int condition);  
int wait\_event\_interruptible (wait\_queue\_head\_t queue, int condition)
  - Sleep until the condition is true

# Blocking I/O

- Wake up at some time later  
wake\_up (wait\_queue\_head\_t \*queue);  
wake\_up\_interruptible (wait\_queue\_head\_t \*queue);  
wake\_up\_sync (wait\_queue\_head\_t \*queue);  
wake\_up\_interruptible\_sync (wait\_queue\_head\_t \*queue);
  - Wake\_up call can cause an immediate reschedule
  - Wake\_up\_sync make awakened processes runnable, but do not reschedule the CPU immediately.

# Blocking I/O

```
DECLARE_WAIT_QUEUE_HEAD(wq);
ssize_t sleepy_read (struct file *filp, char *buf, size_t count, loff_t *pos) {
 printk(KERN_DEBUG "process %i (%s) going to sleep\n",
 current->pid, current->comm);
 interruptible_sleep_on(&wq);
 printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
 return 0; /* EOF */
}
ssize_t sleepy_write (struct file *filp, const char *buf, size_t count,
 loff_t *pos) {
 printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
 current->pid, current->comm);
 wake_up_interruptible(&wq);
 return count; /* succeed, to avoid retrial */
}
```

# Scullpipe

- A variation of scull with blocking IO

```
typedef struct Scull_Pipe {
 wait_queue_head_t inq, outq; /* read and write queues */
 char *buffer, *end; /* begin of buf, end of buf */
 int buffersize; /* used in pointer arithmetic */
 char *rp, *wp; /* where to read, where to write */
 int nreaders, nwriters; /* number of openings for r/w */
 struct fasync_struct *asynch_queue; /* asynchronous readers */
 struct semaphore sem; /* mutual exclusion semaphore */
 devfs_handle_t handle; /* only used if devfs is there */
} Scull_Pipe;
```

# Scullpipe

```
ssize_t scull_p_read (struct file *filp, char *buf, size_t count,
 loff_t *f_pos) {
 Scull_Pipe *dev = filp->private_data;
 if (f_pos != &filp->f_pos) return -ESPIPE;
 if (down_interruptible(&dev->sem)) return -ERESTARTSYS;
 while (dev->rp == dev->wp) { /* nothing to read */
 up(&dev->sem); /* release the lock */
 if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
 if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
 return -ERESTARTSYS;
 if (down_interruptible(&dev->sem)) return -ERESTARTSYS;
 }
 if (dev->wp > dev->rp) /* ok, data is there, return something */
 count = min(count, dev->wp - dev->rp);
 else count = min(count, dev->end - dev->rp);
}
```

# Scullpipe

```
if (copy_to_user(buf, dev->rp, count)) {
 up (&dev->sem);
 return -EFAULT;
}
dev->rp += count;
if (dev->rp == dev->end)
 dev->rp = dev->buffer; /* wrapped */
up (&dev->sem);

/* finally, awake any writers and return */
wake_up_interruptible(&dev->outq);
PDEBUG("\'%s\' did read %li bytes\n", current->comm, (long)count);
return count;
}
```

# Scullpipe

```
static inline int spacefree(Scull_Pipe *dev)
{
 if (dev->rp == dev->wp)
 return dev->buffersize - 1;
 return ((dev->rp + dev->buffersize - dev->wp) % dev-> buffersize) - 1;
}

ssize_t scull_p_write(struct file *filp, const char *buf, size_t count,
 loff_t *f_pos)
{
 Scull_Pipe *dev = filp->private_data;
 if (f_pos != &filp->f_pos) return -ESPIPE;
 if (down_interruptible(&dev->sem))
 return -ERESTARTSYS;
```

# Scullpipe

```
/* Make sure there's space to write */
while (spacefree(dev) == 0) { /* full */
 up(&dev->sem);
 if (filp->f_flags & O_NONBLOCK)
 return -EAGAIN;
 if (wait_event_interruptible(dev->outq, spacefree(dev) > 0))
 return -ERESTARTSYS;
 if (down_interruptible(&dev->sem)) return -ERESTARTSYS;
}
/* ok, space is there, accept something */
count = min(count, spacefree(dev));
if (dev->wp >= dev->rp)
 count = min(count, dev->end - dev->wp);
else /* the write pointer has wrapped, fill up to rp-1 */
 count = min(count, dev->rp - dev->wp - 1);
```

# Scullpipe

```
if (copy_from_user(dev->wp, buf, count)) {
 up (&dev->sem);
 return -EFAULT;
}
dev->wp += count;
if (dev->wp == dev->end)
 dev->wp = dev->buffer; /* wrapped */
up(&dev->sem);
/* finally, awake any reader */
wake_up_interruptible(&dev->inq);
if (dev->async_queue)
 kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
return count;
}
```

# Spinlock

- For SMP (Symmetric MultiProcessing) computer, two processes on two processors can attempt to open the device simultaneously → causes a problem.
  - Can be solved by semaphore, but expensive
  - Spinlock:
    - Never put a process to sleep
    - Keep retrying until the lock is freed.
    - Very little locking overhead
    - Its implementation is empty for a uniprocessor system.
    - Thus, ideal mechanism for small critical sections

# Spinlock

- Defined in <linux/spinlock.h>
- Initialize the spinlock: `spin_lock_init (spinlock_t *lock);`
- Obtain the spinlock: `spin_lock (spinlock_t *lock);`
- Release the spinlock: `spin_unlock (spinlock_t *lock);`
- For `scull_s` device

```
spinlock_t scull_s_lock; // global variable
.....
spin_lock_init(&scull_s_lock); // inside init module
.....
spin_lock(&scull_s_lock); // inside scull_s_open()
.....
scull_s_count++;
spin_unlock (&scull_s_lock);
```

# Access Control

- Access control:
  - Only an unauthorized user allowed to open the device at a time.
- Single-open devices:
  - only one process to open a device at a time

```
Scull_Dev scull_s_device;
int scull_s_count = 0;
spinlock_t scull_s_lock;
int scull_s_open(struct inode *inode, struct file *filp)
{
 Scull_Dev *dev = &scull_s_device; /* device information */
 int num = NUM(inode->i_rdev);
 if (!filp->private_data && num > 0)
 return -ENODEV; /* not devfs: allow 1 device only */
}
```

# Access Control

```
spin_lock(&scull_s_lock);
if (scull_s_count) {
 spin_unlock(&scull_s_lock);
 return -EBUSY; /* already open */
}
scull_s_count++;
spin_unlock(&scull_s_lock);
/* then, everything else is copied from the bare scull device */
if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
 scull_trim(dev);
if (!filp->private_data)
 filp->private_data = dev;
MOD_INC_USE_COUNT;
return 0; /* success */
}
```

# Access Control

```
int scull_s_release(struct inode *inode, struct file *filp)
{
 scull_s_count--; /* release the device */
 MOD_DEC_USE_COUNT;
 return 0;
}
```

# Asynchronous Notification

- Necessity of asynchronous notification
  - Example: a process executes a long computational loop at low priority, but needs to process incoming data as soon as possible using asynchronous notification
- User programs to handle asynchronous notification
  - Tell the kernel who to notify when an input file has new data
    - Specify a process as the owner of the input file (set `filp→f_owner` field)
    - Ex) set the current process as the owner stdin input file:  
`fcntl(STDIN_FILENO, F_SETOWN, getpid());`
  - Enable the asynchronous notification
    - Set the FASYNC flag of the file
    - Ex) set the FASYNC flag of the stdin input file  
`oflags = fcntl(STDIN_FILENO, F_GETFL);`  
`fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);`

# Asynchronous Notification

- When a process receives a SIGIO, it doesn't know which input file has new input; if multiple input files can asynchronously notify, the application needs poll or select.
- Driver's point of view
  - When F\_SETOWN is invoked
    - Kernel set `filp→f_owner`
    - Driver: nothing to do



# The Driver's Point of View

- When `F_SETFL` is executed to turn on `FASYNC`
  - The driver's `fasync()` method is called.

```
int scull_p_fasync(fasync_file fd, struct file *filp, int mode) {
 Scull_Pipe *dev = filp->private_data;
 return fasync_helper (fd, filp, mode, &dev->async_queue); }
```
  - `fasync_helper()` method: add or remove files (depending on the mode) from the queue.
- When data arrives
  - All the registered processes receive `SIGIO` signal

```
kill_fasync(&dev_async_queue, SIGIO, POLL_IN);
```
  - In `scullpipe`, this function is called in `write()` method.
- Remove the file from the queue when closing the file

```
scull_p_fasync(-1, filp, 0);
```

# Task Queues

- A linked list of tasks
- These tasks executed at a later time when the queue is run.
- Each task: a function and an argument
- When a task is run, it calls the function with the argument
- Task queue element

```
struct tq_struct {
 struct tq_struct *next; // linked list
 int sync; // flag to prevent queueing the same task again
 void (*routine) (void *); // function to call
 void *data; // argument to function
}
```

- `task_queue` structure: a pointer to `struct tq_struct`

# Task Queues

- Operations to be performed on task queues

```
DECLARE_TASK_QUEUE (name);
```

```
int queue_task (struct tq_struct *task, task_queue *list);
```

```
 // insert a task into a task queue
```

```
void run_task_queue (task_queue *list);
```

```
 //run a given queue, each entry in the list is executed
```

```
 // no execution order among the tasks in the list
```

- Example)

```
DECLARE_TASK_QUEUE (tq_custom);
```

```
.....
```

```
queue_task (&custom_task, &tq_custom);
```

```
.....
```

```
run_task_queue (&tq_custom);
```

# Task Queues

- Almost certainly not run when the process that queued the task is executing.
- Often run as the result of a “software interrupt.”
- Task queue may run in interrupt mode
- Constraints for interrupt mode execution
  - No access to user space is allowed.
  - The current pointer is not valid.
  - No sleeping or scheduling may be performed
    - May not call schedule or sleep\_on
    - May not call kmalloc (....., GFP\_KERNEL)
    - May not use semaphores

# Tasklets

- Similar to a task queue
- Used in interrupt mode
- Tasklet declaration

DECLARE\_TASKLET (name, function, data)

- Name: tasklet name
- Function: the function to be called when a tasklet is run
- Data: the argument of the function

DECLARE\_TASKLET\_DISABLED (name, function, data)

- Not executed until enabled at some later time

- Example)

```
void jiq_print_tasklet (unsigned long);
```

```
DECLARE_TASKLET (jiq_tasklet, jiq_print_tasklet, (unsigned long)
&jiq_data);
```

- To schedule the tasklet to run  
tasklet\_schedule (&jiq\_tasklet);

# Tasklets

- Other functions for tasklets

```
void tasklet_disable (struct tasklet_struct *t);
```

```
void tasklet_enable (struct tasklet_struct *t);
```

```
void tasklet_kill (struct tasklet_struct *t);
```

# Kernel Timers

- Timer structure in <linux/timer.h>

```
struct timer_list {
 struct timer_list *next;
 struct timer_list *prev;
 unsigned long expires; // the timeout
 unsigned long data; // argument to function()
 void (*function) (unsigned long); // function to call
 volatile int running;
};
```

- timer->function() will run when timer->expires <= jiffies

# Kernel Timers

- Functions for timers

```
void init_timer (struct timer_list *timer);
void add_timer (struct timer_list *timer);
int mod_timer (struct timer_list *timer, unsigned
long expires)
 ▪ modify the timer expires
int del_timer (struct timer_list *timer);
 ▪ Remove a timer before expires
int del_timer_sync (struct timer_list *timer);
 ▪ Guarantees the timer function is not running
```

# kmalloc

- Allocation flag (the second argument of `kmalloc()`): controls the behavior of `kmalloc`  
`scullc_devices`  
    = `kmalloc(scullc_devs * sizeof(ScullC_Dev), GFP_KERNEL);`
- Two most widely used flags
  - `GFP_KERNEL`: can put the current process to sleep waiting for a page when called in low-memory situations
  - `GFP_ATOMIC`:
    - Cannot put to sleep
    - Can use even the last free page
    - If the last page does not exist, the `kmalloc()` fails
- Other flags (defined in `<linux/mm.h>`)\*
  - `GFP_BUFFER`, `GFP_USER`, `GFP_HIGHUSER`, `__GFP_DMA`, `GFP_HIGHMEM`

# Use of the `ioctl` Commands

- A user program may want to use a system call other than `read()` or `write()`
- With `ioctl()`, a driver can perform various types of device specific hardware control that cannot be performed by `read/write` operations

```
int main() {
 int quantum;

 ioctl (fd, SCULL_IOCSEQQUANTUM, &quantum);
 ioctl (fd, SCULL_IOCTLQUANTUM, quantum);
 ioctl (fd, SCULL_IOCQQUANTUM, &quantum);
 quantum=ioctl (fd, SCULL_IOCQQUANTUM);
 ioctl (fd, SCULL_IOCXQUANTUM, &quantum);
 quantum=ioctl (fd, SCULL_IOCQQUANTUM);
}
```

# ioctl method

- User space call to the ioctl function() format  
`int ioctl (int fd, int cmd, ...);`
  - ↑
  - single optional argument that depends on the second argument
  - no type checking at compile-time
- Driver method  
`int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);`
  - inode and filp are derived from fd
  - cmd is passed unchanged
  - arg is passed in the form of unsigned long

# Choosing the ioctl Commands

- A command is represented by a unique number
- Old Linux convention
  - 16-bit number: top eight bit for the device-specific number, bottom eight bit for the unique number in a device
  - Ex) Assume '0x6b' is the device-specific magic number  
`#define SCULL_IOCTL1 0x6b01`  
`#define SCULL_IOCTL2 0x6b02`
- New convention
  - Type: device-specific magic number
  - Number: sequential number used in a device
  - Direction: `_IOC_NONE` (no data transfer), `_IOC_READ`, `IOC_WRITE`, and `IOC_READ | _IOC_WRITE`
  - Size: user data size; if larger data structures needed, just ignore this field.

# User-space Access

- Ensure the user address is valid and currently in memory  
→ need to check every user-space address
- Function `access_ok()`  
`int access_ok (int type, const void *addr, unsigned long size);`
  - type: `VERIFY_READ` or `VERIFY_WRITE`
  - addr: user-space address
  - size: byte count
  - Return value: 1 for success and 0 for failure
- In `scull` example,  
If `(IOC_DIR (cmd) & _IOC_READ)`  
`err = !access_ok (VERIFY_WRITE, (void *)arg, _IOC_SIZE (cmd));`  
else if `(IOC_DIR (cmd) & _IOC_WRITE)`  
`err = !access_ok (VERIFY_READ, (void *)arg, _IOC_SIZE (cmd));`  
If `(err)` return `-EFAULT;`

# User-space Access

- Function `put_user()/get_user()`
  - Optimized data transfer for one, two, and four bytes
    - Better than `copy_to_user()/copy_from_user()`
  - Defined in `<asm/uaccess.h>`
  - `put_user (datum, ptr)/__put_user (datum, ptr)`
    - “datum” is to be transferred to the user area pointed by “ptr”.
    - `put_user()` checks to ensure that the process can write to the given memory address
    - `__put_user()` does less checking; thus needs to verify with `access_ok()` before calling this function.
  - `get_user (local, ptr)/__get_user (local, ptr)`
    - Similar to `put_user()/__put_user()`
    - Data transfer direction is opposite

# Restricted Operations

- Driver checks if a user can perform the requested operation.
- Capable function (defined in <sys/sched.h>)  
int capable (int capability);
- Example:  
if (! capable (CAP\_SYS\_ADMIN)) return -EPERM;  
ret = \_\_get\_user (scull\_quantum, (int \*)arg);
- Other capability levels (in <linux/capability.h>):
  - CAP\_DAC\_OVERRIDE
  - CAP\_NET\_ADMIN
  - CAP\_SYS\_MODULE
  - CAP\_SYS\_RAW
  - CAP\_SYS\_ADMIN
  - CAP\_SYS\_TTY\_CONFIG

# ioctl Commands

```
switch(cmd) {
 case SCULL_IOCTLRESET:
 scull_quantum = SCULL_QUANTUM;
 scull_qset = SCULL_QSET;
 break;
 case SCULL_IOCTLSCQUANTUM: /* Set: arg points to the value */
 if (! capable (CAP_SYS_ADMIN))
 return -EPERM;
 ret = __get_user(scull_quantum, (int *)arg);
 break;
 case SCULL_IOCTLQQUANTUM: /* Tell: arg is the value */
 if (! capable (CAP_SYS_ADMIN))
 return -EPERM;
 scull_quantum = arg;
 break;
```



# ioctl Commands

```
case SCULL_IOCTLGQUANTUM: /* Get: arg is pointer to result */
 ret = __put_user(scull_quantum, (int *)arg);
 break;
case SCULL_IOCTLQQUANTUM: /* Query: return it */
 return scull_quantum;
case SCULL_IOCTLXQUANTUM:
 if (!capable(CAP_SYS_ADMIN))
 return -EPERM;
 tmp = scull_quantum;
 ret = __get_user(scull_quantum, (int *)arg);
 if (ret == 0)
 ret = __put_user(tmp, (int *)arg);
 break;
default:
 return -ENOTTY;
```

# I/O Ports and I/O Memory

- Peripheral control: writing and reading its registers.
- I/O memory: memory-mapped I/O
- I/O port: separate I/O address space
  - Separate read/write electrical lines for I/O ports
  - Special CPU instructions to access I/O ports
- I/O memory approach is often preferred
  - No need of special-purpose processor instructions
  - CPU can access memory efficiently
  - Compiler has freedom for optimizing memory access than I/O access

# I/O Memory

- Allocate I/O memory region before using it

```
int check_mem_region (unsigned long start, unsigned long len);
void request_mem_region (unsigned long start, unsigned long len,
char *name);
void release_mem_region (unsigned long start, unsigned long len);
```
- Avoid direct access of the memory region, but use the following functions
  - address can be either integer or pointer

```
unsigned readb (address); // read 8-bit
unsigned readw (address); // read 16-bit
unsigned readl (address); // read 32-bit
unsigned writeb (unsigned value, address); // write 8-bit
unsigned writew (unsigned value, address); // write 16-bit
unsigned writel (unsigned value, address); // write 32-bit
```

# I/O Memory

- Code fragment writing to a memory location

```
while (count--) {
 writeb (*(ptr++), address);
 wmb();
}
```
- Other functions

```
memset_io (address, value, count); // set memory
contents
memcpy_fromio (dest, source, num); // copy from io
memory
memcpy_toio (dest, source, num); // copy to io
memory
```

# Software-Mapped Memory

- Most common HW/SW arrangement of I/O memory
    - Devices at well-known physical addresses
    - CPU has no predefined virtual address to access them
    - Assign a virtual address to the device with `ioremap()`
- ```
void *ioremap (unsigned long phys_addr, unsigned long size);  
void *ioremap_nocache (unsigned long phys_addr, unsigned long size);  
void iounmap (void *addr);
```
- `ioremap_nocache`: remap for non-cacheable io memory.

Software-Mapped Memory

```
#define ISA_BASE 0xA0000  
#define ISA_MAX 0x100000 /* for general memory access */  
static void *io_base;  
silly_init() { .....  
    io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);  
    ..... }  
silly_read(....., size_t count, loff_t *f_pos) { .....  
    unsigned long isa_addr = ISA_BASE + *f_pos;  
    if (isa_addr + count > ISA_MAX) /* range: 0xA0000-0x100000 */  
        count = ISA_MAX - isa_addr;  
    .....  
    add = io_base + (isa_addr - ISA_BASE);  
    .....  
    while (count) {  
        *ptr = readb(add);  
        add++; count--; ptr++;
```

I/O Memory

- Difference between I/O registers and memory
 - Memory: no side effect → many optimizations possible
 - Data caching
 - Access reordering
 - I/O registers: side effect → no caching or reordering allowed
 - Solution to avoid side effect
 - Data caching: Linux is configured to disable any caching when accessing I/O regions
 - Access reordering:
 - place a memory barrier to prevent reordering
 - Store to memory all values modified and resident in CPU registers.
- ```
#include <linux/kernel.h>
void barrier (void)
```
- Barriers reduce performance → used only when necessary

# IO Memory

- Guarantee ordering of both reads and writes.

```
#include <asm/system.h>
void rmb (void)
```
- Guarantee ordering of writes.

```
void wmb (void)
```
- Guarantee ordering of both reads and write: `void mb (void)`
- Example)
  - program to set control registers and then start.
  - make sure all registers are updated before start execution

```
writel (dev->registers.addr, io_destination_address);
writel (dev->registers.size, io_size);
writel (dev->registers.operation, DEV_READ);
wmb();
writel (dev->registers.control, DEV_GO);
```

# Using I/O Ports

- To allocate I/O ports, request it first

```
#include <linux/ioport.h>
int check_region (unsigned long start, unsigned long len);
struct resource *request_region (unsigned long start, unsigned long len,
char *name);
void release_region (unsigned long start, unsigned long len);
```
- Functions to access I/O ports
  - Read or write byte ports

```
unsigned inb (unsigned port);
void outb (unsigned char byte, unsigned port);
```
  - Read or write 16-bit ports

```
unsigned inw (unsigned port);
void outw (unsigned short word, unsigned port);
```
  - Read or write 32-bit ports

```
unsigned inl (unsigned port);
void outl (unsigned longword, unsigned port);
```



# A Sample Driver

- Driver: short (Simple Hardware Operations and Raw Tests)
  - Read and write eight-bit ports
  - By default, it accesses the address at 0x378
  - Each device (w/ unique minor number) accesses a different port.
    - /dev/short0 accesses the eight-bit port at 0x378
    - /dev/short1 accesses the eight-bit port at 0x379, and so on
- Short must be able to allocate the I/O region.
  - If you get a “can’t get I/O address” error, check “/proc/ioports” to see if any other devices hold the address.
  - can configure at load time to use I/O memory and change the base address

Example) `./short_load use_mem=1 base=0b7fffc0`



# A Sample Driver

```
static int use_mem = 0; /* default is I/O-mapped */
MODULE_PARM(use_mem, "i");
static unsigned long base = 0x378;
unsigned long short_base = 0;
MODULE_PARM(base, "I");
.....
enum short_modes {SHORT_DEFAULT=0, SHORT_PAUSE,
 SHORT_STRING, SHORT_MEMORY};
.....
ssize_t do_short_write (struct inode *inode, struct file *filp, const char
 *buf, size_t count, loff_t *f_pos)
{
 int retval = count;
 unsigned long address = short_base + (MINOR(inode->i_rdev)&0x0f);
 int mode = (MINOR(inode->i_rdev)&0x70) >> 4;
```

# A Sample Driver

```
unsigned char *kbuf=kmalloc(count, GFP_KERNEL), *ptr;
if (!kbuf) return -ENOMEM;
if (copy_from_user(kbuf, buf, count)) return -EFAULT;
ptr=kbuf;
if (use_mem) mode = SHORT_MEMORY;
switch(mode) {
 case SHORT_PAUSE:
 while (count--) {
 outb_p(*(ptr++), address);
 wmb();
 } break;
 case SHORT_STRING:
 outsb(address, ptr, count);
 wmb();
 break;
```

# A Sample Driver

```
case SHORT_DEFAULT:
 while (count--) {
 outb(*(ptr++), address);
 wmb();
 }
 break;
case SHORT_MEMORY:
 while (count--) {
 writeb(*(ptr++), address);
 wmb();
 }
 break;
default: /* no more modes defined by now */
 retval = -EINVAL;
 break;
```

# Interrupt Handler (Routine)

- Registration of an interrupt handler:

```
int request_irq (unsigned int irq, void (*handler) (int, void
*, struct pt_regs *), unsigned long flags, const char
*dev_name, void *dev_id);
```

- irq: interrupt number
- \*handler : pointer to the handling function
- flags: bit mask option for interrupt management
  - SA\_INTERRUPT: fast interrupt handler (executed with interrupts disabled)
  - SA\_SHIRQ: interrupt can be shared between devices
  - SA\_SAMPLE\_RANDOM: set when interrupts are generated randomly.
  - \*dev\_name: string that appears in /proc/interrupts to show the owner of the interrupt

# Interrupt Handler

- `*dev_id`:
  - pointer used for shared interrupt lines.
  - Identify which device is interrupting.
  - passed to an argument of the handler
- Example

```
int short_init(void)
{

 result = request_irq(short_irq, short_interrupt,
 SA_INTERRUPT, "short", NULL);

}
```
- Release of the handler

```
void free_irq (unsigned int irq, void *dev_id);
```

# Interrupt Handler

- Interrupt handler installation location
  - Installation at driver initialization: the device may hold the interrupt line even never used.
  - Installation at device open: may share the same interrupt line by many devices → correct place
- The `/proc` interface
  - `/proc/interrupts`: snapshot of interrupts for several days of uptime
    - Show how many interrupts are delivered.
  - `/proc/stat`: the number of interrupts received since system boot

```
intr 884865 695557 4527 0 3109 4907 112759
total number irq0 irq1 irq2 irq3 irq4 irq5
```



# Interrupt Handler

- Ordinary C program
  - Feedback to its device to inform the reception of the interrupt; clear interrupt pending bit
  - Read or write data according to the interrupt request
  - Wake up a sleeping process that waits for the interrupt
  - Executes in a minimum of time
  - If long computation needs, use tasklet to schedule at a safer time
- Restrictions
  - Cannot transfer data to/from user space
  - Cannot sleep
  - Cannot allocate memory with anything other than GFP\_ATOMIC
  - Cannot lock a semaphore
  - Cannot call a schedule

# Interrupt Handler

- Example)

```
void short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
 struct timeval tv;
 int written;
 do_gettimeofday(&tv);
 written = sprintf((char *)short_head,"%08u.%06u\n",
 (int)(tv.tv_sec % 100000000),
 (int)(tv.tv_usec));
 // increment the short_head
 short_incr_bp(&short_head, written);
 wake_up_interruptible(&short_queue); /* awake any
 reading process */
```

# Interrupt Handler

```
ssize_t short_i_write (struct file *filp, const char *buf, size_t count,
 loff_t *f_pos) {
 int written = 0, odd = *f_pos & 1;
 unsigned long address = short_base;
 if (use_mem) {
 while (written < count)
 writeb(0xff * ((++written + odd) & 1), address);
 } else {
 while (written < count)
 outb(0xff * ((++written + odd) & 1), address);
 }
 *f_pos += count;
 return written;
}
```

# Interrupt Handler

```
int request_irq (unsigned int irq, void (*handler) (int, void *, struct pt_regs
*), unsigned long flags, const char *dev_name, void *dev_id);
```

- void \*dev\_id: passed to an argument of the handler
- Example)

```
static void sample_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
 struct sample_dev *dev = dev_id;
 // now 'dev' points to the right hardware

}

static void sample_open (struct inode *inode, struct file *filp) {
 struct sample_dev *dev = hwinfo+MINOR(inode->i_rdev);
 request_irq (dev->irq, sample_interrupt, 0, "sample", dev);

}
```

# Interrupt Handler

- Update the mask for the specified interrupt in the programmable interrupt controller (PIC)  
void disable\_irq (int irq);  
void disable\_irq\_nosync (int irq);  
void enable\_irq (int irq);
- This call can be nested: if disable\_irq is called twice in succession, two enable\_irq calls are necessary to enable the interrupt.
- disable\_irq() waits for the completion of currently executing interrupt handler, while disable\_irq\_nosync() does not.

# Bottom-Half Processing

- Main issues for interrupt handling:
  - How to perform longish tasks within a handler.
  - Interrupt handler must not keep other interrupts blocked for long
  - Solution: splitting the interrupt handler into two halves
    - Top half: the routine that actually responds to the interrupt; the one registers with request\_irq.
    - Bottom half: the routine scheduled by the top half to be executed later.
- Bottom half
  - All interrupts are enabled during execution of the bottom half.
  - All restrictions that apply to interrupt handlers also apply to bottom halves: no sleep, no user space access, no scheduler invocation.

# Bottom-Half Processing

- Tasklets in short driver

```
void short_do_tasklet (unsigned long);
```

```
DECLARE_TASKLET (short_tasklet, short_do_tasklet, 0);
```

- Top half

```
void short_tl_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
```

```
{
```

```
do_gettimeofday((struct timeval *) tv_head); /* cast to stop
'volatile' warning */
```

```
short_incr_tv(&tv_head);
```

```
tasklet_schedule(&short_tasklet);
```

```
short_bh_count++; /* record that an interrupt arrived */
```

```
}
```

# Bottom-Half Processing

- Bottom half

```
void short_do_tasklet (unsigned long unused)
```

```
{
```

```
int savecount = short_bh_count, written;
```

```
short_bh_count = 0;
```

```
written = sprintf((char *)short_head, "bh after %6i\n", savecount);
```

```
short_incr_bp(&short_head, written);
```

```
do {
```

```
written = sprintf((char *)short_head, "%08u.%06u\n",
```

```
(int)(tv_tail->tv_sec % 10000000),
```

```
(int)(tv_tail->tv_usec));
```

```
short_incr_bp(&short_head, written);
```

```
short_incr_tv(&tv_tail);
```

```
} while (tv_tail != tv_head);
```

```
wake_up_interruptible(&short_queue);
```

# Interrupt Sharing

- IRQ conflict: # of interrupt lines < # of devices  
→ interrupt line sharing necessary
- Installing a shared Handler  
`int request_irq (unsigned int irq, void (*handler) (int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);`
- Set the flag SA\_SHIRQ
- `dev_id` must be unique.
- The kernel invokes every handler registered for the given interrupt passing its own `dev_id`.
- The shared handler should quickly exit when its own device has not interrupted.
- Each driver releases the handler with its own `dev_id`.

# Interrupt Sharing

```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs
 *regs)
{
 int value, written;
 struct timeval tv;
 /* If it wasn't short, return immediately */
 value = inb(short_base);
 if (!(value & 0x80)) return;
 /* clear the interrupting bit */
 outb(value & 0x7F, short_base);
 /* the rest is unchanged */

```

# Linux Block Drivers

## Block Drivers

- Classic block device: a disk drive
- The block interface is messier than the char driver
  - The block interface has been the core of every Linux version since the first
  - Trade-off with performance
- sbull (Simple Block Utility for Loading Localities) and spull are used for this chapter
- Functions for registration/unregistration

```
#include <linux/fs.h>
```

```
int register_blkdev (unsigned int major, const char *name, struct
 block_device_operations *bdops);
```

```
int unregister_blkdev (unsigned int major, const char *name);
```

- Global array blk\_dev[] is updated when register\_blkdev() is called.
- Dynamic registration is also possible: similar to scull registration

# Registering the Driver

- Data structure

```
struct block_device_operations {
 int (*open) (struct inode *inode, struct file *filp);
 int (*release) (struct inode *inode, struct file *filp);
 int (*ioctl) (struct inode *inode, struct file *filp, unsigned command,
 unsigned long argument);
 int (*check_media_change) (kdev_t dev);
 int (*revalidate) (kdev_t dev);
};
struct block_device_operations sbull_bdops = {
 open: sbull_open,
 release: sbull_release,
 ioctl: sbull_ioctl,
 check_media_change: sbull_check_change,
 revalidate: sbull_revalidate,
```

# Registering the Driver

- Request method and request queue
  - When the kernel schedules a data transfer, it queues the request in a list, ordered to maximize performance.
  - The queue of requests is passed to the driver's request function.
- Initialize/cleanup the queue of I/O operations

```
#include <linux/blkdev.h>
```

```
blk_init_queue (request_queue_t *queue, request_fn_proc *request);
```

```
blk_cleanup_queue (request_queue_t *queue);
```

- Inside sbull

```
blk_init_queue (BLK_DEFAULT_QUEUE (major), sbull_request);
```

- BLK\_DEFAULT\_QUEUE(major): the default request queue

# Registering the Driver

- Global arrays holding information about the driver
  - In sbull, these are initialized at module initialization
  - `int blk_size[][]`:
    - indexed by the major and minor numbers
    - The size of each device in kilobytes
  - `int blksize_size[][]`:
    - Indexed by the major and minor numbers
    - The size of the block used by each device in bytes
  - `int hardsect_size[][]`:
    - Indexed by the major and minor numbers
    - Disk sector size; default is 512 bytes.
  - `int read_ahead[]; int max_readahead[]`:
    - The number of sectors to be read in advance for a file read
  - `int max_sectors[][]`:
    - The maximum size of a single request;

# Registering the Driver

- Default values in sbull
  - `size=2048, blksize=1024, hardsect=512, rahead=2`
- Register “disk” device: set up the partition table  
for (`i=0; i<sbull_devs; i++`)  
`register_disk (NULL, MKDEV(major,i), 1,`  
`&sbull_bdops, sbull_size << 1);`
- Cleanup function in sbull  
for (`i=0; i<sbull_devs; i++`)  
`fsync_dev (MKDEV(sbull_major, 1)); // flush the`  
`device`  
`unregister_blkdev (major, “sbull”);`  
`blk_cleanup_queue (BLK_DEFAULT_QUEUE(major));`



# Registering the Driver

- Cleanup function in sbull (continued)

```
// clean up the global arrays
read_ahead [major] = 0;
kfree (blk_size[major]);
blk_size[major] = NULL;
kfree (blksize_size[major]);
blksize_size [major] = NULL;
kfree (hardsect_size[major]);
hardsect_size [major] = NULL;
```

# The Header File blk.h

- blk.h defines codes commonly used in block drivers.
- Some symbols need to be predefined before this file
- MAJOR\_NR
  - Used to access arrays
  - Should define it before including this header or
  - #define it to the variable holding the number

```
#define MAJOR_NR sbull_major;
static int sbull_major;
.....
#include <linux/blk.h>
```
- DEVICE\_NAME: name of the device

```
#define DEVICE_NAME "sbull"
```
- DEVICE\_NR (kdev\_t device): used to derive the device number from kdev\_t

```
#define DEVICE_NR (device) MINOR(device)
```

# The Header File blk.h

- `DEVICE_INTR`: pointer to the bottom half  
`#define DEVICE_INTR sbull_intrptr`
- `DEVICE_REQUEST`: the name of the request function used by the driver  
`#define DEVICE_REQUEST sbull_request`

# Handling Requests

- The request queue  
`void request_fn (request_queue_t *queue);`
  - Tasks of the request function
    - Check the validity of the request; performed by macro `INIT_REQUEST`
    - Perform the actual data transfer; use `CURRENT` to retrieve the details of the current request;  
`CURRENT->cmd, CURRENT->sector,,,,`
    - Clean up the request just processed; performed by `end_request`
      - Manages the request queue
      - Wakes up processes waiting on the I/O operations
      - Manages the `CURRENT` variable.
      - Passes argument '1' for success and '0' for failure.
    - Look back to the beginning to process the next request.

# Handling Requests

- Possible minimal request function

```
void sbull_request (request_queue_t *q)
{
 while (1) {
 INIT_REQUEST;
 printk (“<1>request %p: %i sec %li (nr, %li)\n”, CURRENT,
 CURRENT->cmd, CURRENT->sector,
 CURRENT->current_nr_sectors);
 end_request (1);
 }
}
```

- INIT\_REQUEST returns when the request queue is empty → while ends.
- CURRENT always points to the request to be processed.
- The request function must not sleep.

# Handling Requests

- Performing the Actual Data Transfer

- The fields of struct request
  - kdev\_t rq\_dev; the device accessed by the request
  - int cmd; the operation to be performed; either READ or WRITE
  - unsigned long sector; the number of the first sector to be transferred in this request
  - unsigned long current\_nr\_sectors; the number of sectors to transfer
  - char \*buffer: the area in the buffer cache
  - struct buffer\_head \*bh; the first buffer in the list

# Handling Requests

```
void sbull_request(request_queue_t *q) {
 Sbull_Dev *device; int status;
 while(1) {
 INIT_REQUEST; /* returns when queue is empty */
 /* Which "device" are we using? */
 device = sbull_locate_device (CURRENT);
 if (device == NULL) {
 end_request(0);
 continue;
 }
 spin_lock(&device->lock);
 status = sbull_transfer(device, CURRENT);
 spin_unlock(&device->lock);
 end_request(status);
 }
}
```

# Handling Requests

```
static Sbull_Dev *sbull_locate_device(const struct request *req)
{
 int devno;
 Sbull_Dev *device;
 /* Check if the minor number is in range */
 devno = DEVICE_NR(req->rq_dev);
 if (devno >= sbull_devs) {
 static int count = 0;
 if (count++ < 5) /* print the message at most five times */
 printk(KERN_WARNING "sbull:request unknown device\n");
 return NULL;
 }
 device = sbull_devices + devno; /* out of our device array */
 return device;
}
```

# Handling Requests

```
static int sbull_transfer(Sbull_Dev *device, const struct request *req)
{
 int size; u8 *ptr;

 ptr = device->data + req->sector * sbull_hardsect;
 size = req->current_nr_sectors * sbull_hardsect;
 /* Make sure that the transfer fits within the device. */
 if (ptr + size > device->data + sbull_blksize*sbull_size) {
 static int count = 0;
 if (count++ < 5)
 printk(KERN_WARNING "sbull: request past end of device\n");
 return 0;
 }
}
```

# Handling Requests

```
/* Looks good, do the transfer. */
switch(req->cmd) {
 case READ:
 memcpy(req->buffer, ptr, size); /* from sbull to buffer */
 return 1;
 case WRITE:
 memcpy(ptr, req->buffer, size); /* from buffer to sbull */
 return 1;
 default:
 /* can't happen */
 return 0;
}
```

# Handling Requests

- For performance improvement → need to know the details of how the I/O request queue works
- The I/O Request Queue
  - The performance depends on how to manage this queue.
  - With disks, the data transfer time is small while the head positioning time is large.
  - Need to cluster the requests → use the algorithm for elevator; move the disk header in one direction and processes all requests, and then move the header in the other direction.
- The request structure and the buffer cache
  - Buffer cache: a memory region that holds copies of blocks stored on disk.
  - To keep track of the buffer cache, `buffer_head` structure is used

# Handling Requests

- The request structure and the buffer cache (continued)
  - Main fields of `buffer_head` structure
    - `Char *b_data;` the actual data block
    - `struct buffer_head *b_reqnext;` the pointer to the next `buffer_head`
  - Main fields of request structure
    - `struct buffer_head bh;` points to the first `buffer_head`
    - `char *buffer;` points to the first `buffer_head->b_data`
  - For performance optimization, all of the `buffer_heads` attached to a single request belongs to an adjacent group of blocks (clustering) on the disk.

# Handling Requests

- The I/O request lock
  - The request queue is main subject to the race conditions
  - All request queues are protected with a single global spinlock called `io_request_lock`
  - The kernel calls the request function with the `io_request_lock`
- Clustered Requests
  - Clustering: joining together requests that operate on adjacent blocks on the disk.
  - A driver can improve performance by explicitly acting on clustering.

# Multiqueue Block Drivers

- A driver can set up independent queues for each device.
- In `sbull`
  - A block driver must define its own request queues
    - `Sbull_Dev` structure includes

```
request_queue_t queue;
int busy; // flag to protect the queue
```
    - Request queues must be initialized

```
for (I=0; I<sbull_devs; I++) {
 blk_init_queue (&sbull_devices[I].queue, sbull_request);
 blk_queue_headactive (&sbull_devices[I].queue, 0);
 // marks the queue as not having active heads
}
blk_dev[major].queue = sbull_find_queue;
// sbull_find_queue is the function to find the request queue for
each
// device
```

# Multiqueue Block Drivers

```
request_queue_t *sbull_find_queue (kdev_t device) {
 int devno = DEVICE_NR (device);
 if (devno >= sbull_devs) { // unknown device

 }
 return &sbull_devices[devno].queue;
}
```

- Sbull request function: manipulates the request queue directly.

```
void sbull_request (request_queue_t *q) {
 Sbull_Dev *device;
 struct request *req;
 int status;
 // find our device
 device = sbull_locate_device (blkdev_entry_next_request
 (&q->queue_head));
 if (device->busy) return;
 device->busy = 1;
```

# Multiqueue Block Drivers

```
// process requests in the queue
while (! list_empty (&q->queue_head)) //replacing INIT_REQUEST
{
 // Pull the next request off the list
 req = blkdev_entry_next_request (&q->queue_head);
 blkdev_dequeue_request (req); // remove a request from queue
 spin_unlock_irq (&io_request_lock);
 spin_lock (&device->lock);

 // process data transfer
 do {
 status = sbull_transfer (device, req);
 } while (end_that_request_first (req, status, DEVICE_NAME);
 spin_unlock (&device->lock);
 spin_lock_irq (&io_request_lock);
 end_that_request_last (req); // clean up the request as a
 whole
```



# Multiqueue Block Drivers

- Busy flag used to prevent multiple invocations of `sbulld_request`.
- Better to release one lock before acquiring another to avoid possible deadlock.
- Clean up all of their queues at module removal time  
for (`i=0, i<sbulld_devs, i++`)  
`blk_cleanup_queue (&sbulld_devices[i].queue);`  
`blk_dev[major].queue = NULL;`

# Additional Features

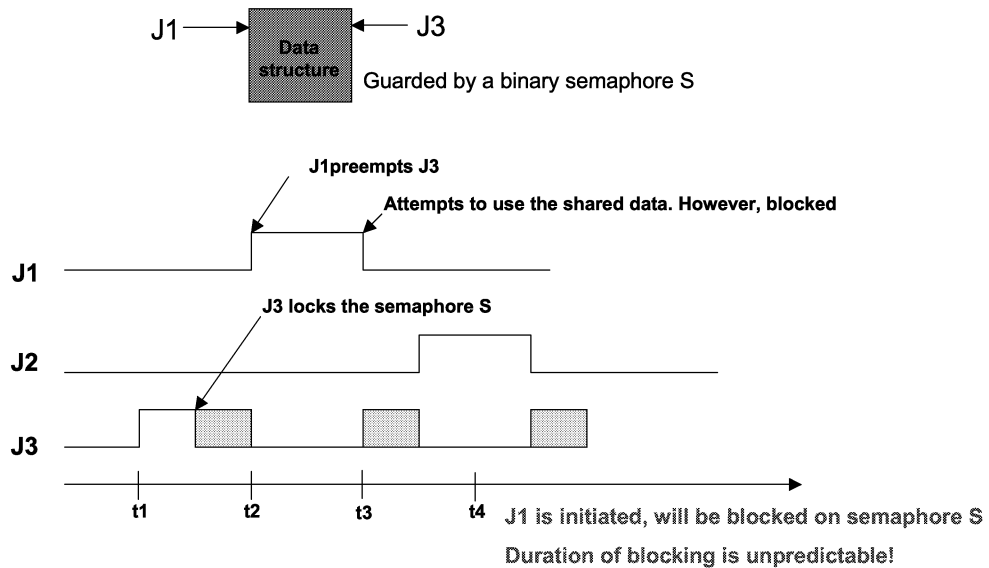
- Support of `ioctl` method:
  - block drivers share many `ioctl` commands
  - Kernel 2.4 provides a function, `blk_ioctl`, for the common commands.
- Support of the removable devices: the last two file operations in the `block_device_operations`
- Support of partitionable devices
- Support of interrupt-driven block drivers

# Real-Time Synchronization

# Priority Inversion Problem

- A resource shared between high and low priority tasks.
- During the time the low priority task locks the shared resource, the high priority task will be blocked if it also tries to access the resource.
- The solution is priority inheritance which must be implemented in the synchronization primitives

# Blocking



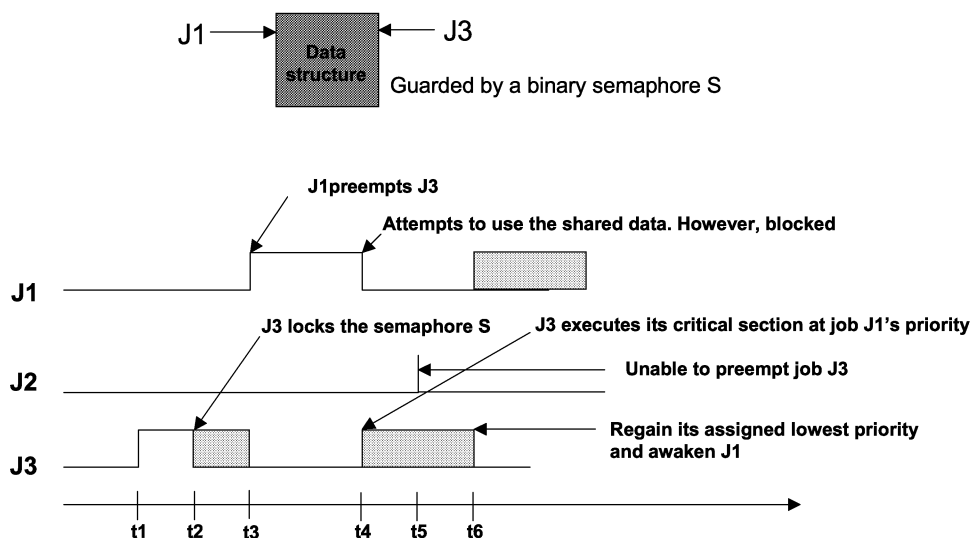
# Real-Time Synchronization techniques

- Priority Inheritance Protocol
  - Basic Priority Inheritance Protocol (BPI)
    - Indefinite priority inversion prevention
    - Deadlock occurs
    - Long blocking time occurs
  - Priority Ceiling Protocol (PCP)
    - Deadlock prevention
    - Long blocking time prevention

# Basic Priority Inheritance Protocol

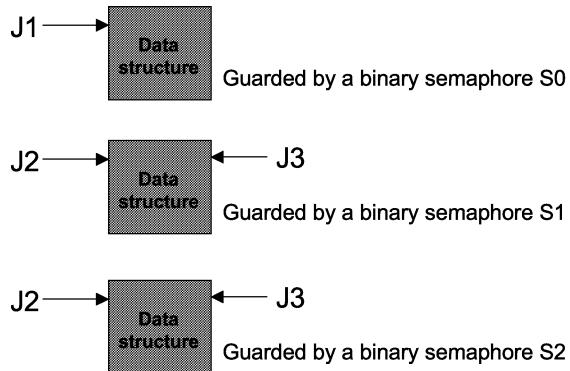
- Basic priority inheritance protocol
  - When a job J blocks one or more higher priority jobs,
    - It ignores its original priority assignment
    - Executes its critical section at the highest priority level of all the jobs it blocks
  - After exiting its critical section,
    - Job J returns its original priority level

# Basic Priority Inheritance Protocol



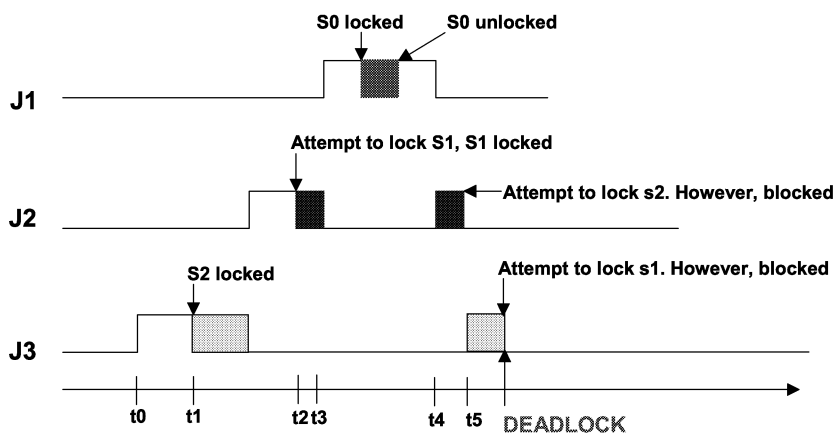
# Deadlock

- $J1 = \{ \dots, P(S0), \dots, V(S0), \dots \}$
- $J2 = \{ \dots, P(S1), \dots, P(S2), \dots, V(S2), \dots, V(S1), \dots \}$
- $J3 = \{ \dots, P(S2), \dots, P(S1), \dots, V(S1), \dots, V(S2), \dots \}$



# Deadlock (cont'd)

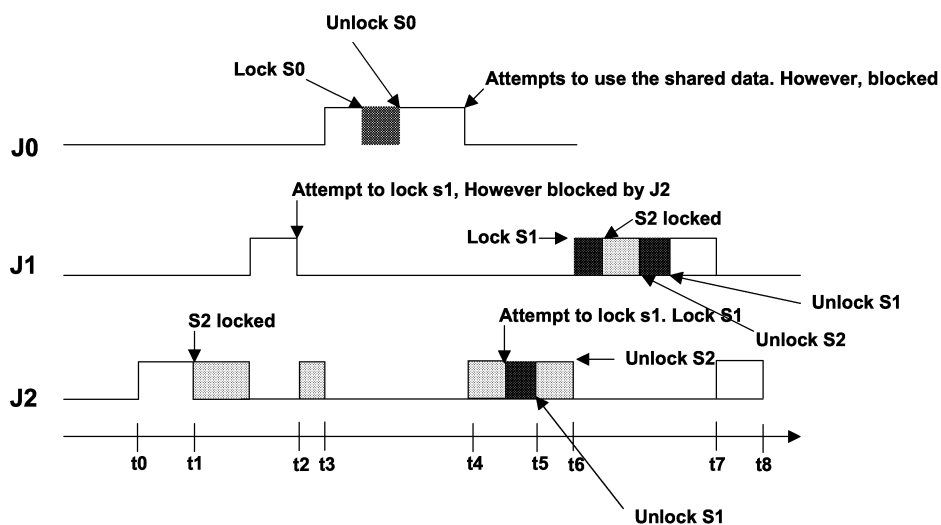
- The problem of the basic priority inheritance protocol



# The Priority Ceiling Protocol

- The goal of this protocol is to prevent the formation of deadlocks and of chained blocking
  - When a job J preempts the critical section of **another job** and executes its own critical section z
  - The priority at which this new critical section z will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections
- $J_0 = \{ \dots, P(S_0), \dots, V(S_0), \dots \}$
- $J_1 = \{ \dots, P(S_1), \dots, P(S_2), \dots, V(S_2), \dots, V(S_1), \dots \}$
- $J_2 = \{ \dots, P(S_2), \dots, P(S_1), \dots, V(S_1), \dots, V(S_2), \dots \}$
- We define the *priority ceiling* of a semaphore as the priority of the highest priority job that may lock this semaphore
- Thus, the priority ceiling of both semaphores S1 and S2 are equal to the priority of job J1

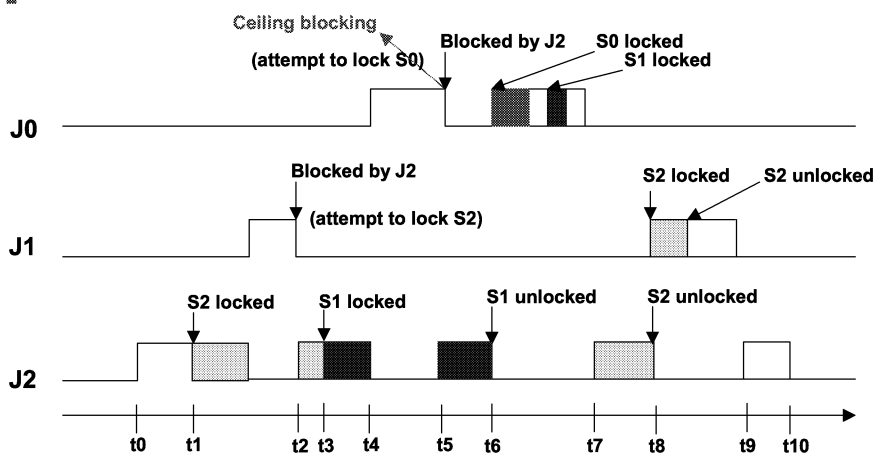
# The Priority Ceiling Protocol



# The Priority Ceiling Protocol

- $J_0 = \{ \dots, P(S_0), \dots, V(S_0), \dots, P(S_1), \dots, V(S_1), \dots \}$
- $J_1 = \{ \dots, P(S_2), \dots, V(S_2), \dots \}$
- $J_2 = \{ \dots, P(S_2), \dots, P(S_1), \dots, V(S_1), \dots, V(S_2), \dots \}$
- $S_0, S_1 \Rightarrow P_0$
- $S_2 \Rightarrow P_1$

# The Priority Ceiling Protocol



Ceiling blocking is needed for the avoidance of deadlock and of chained blocking

# References

- A. Rubini, J. Corbet, Linux Device Drivers, 2<sup>nd</sup> ed.
- D.E. Simon, Embedded Software Architecture
- R. Stones and N. Matthew, Beginning Linux Programming, 2nd ed
- 조유근외, Kernel programming
- And so on.



Sponsored by:

