dit
UPM

Concurrencia

Juan Antonio de la Puente
DIT/UPM

Objetivos

- Repasaremos los principios de la programación concurrente
- Analizaremos la distintas alternativas de ejecución de procesos concurrentes
- Veremos cómo crear procesos concurrentes en Ada y C/POSIX

Índice

Concurrencia

- Los sistemas de tiempo real son concurrentes por naturaleza
- Las actividades del mundo exterior evolucionan simultáneamente

Veremos cómo expresar de forma abstracta la ejecución de actividades potencialmente paralelas

- Los detalles de implementación son independientes de la programación concurrente

Ejemplo: paralelismo en E/S

Sistemas concurrentes

Un sistema concurrente se compone de un conjunto de procesos secuenciales autónomos que se ejecutan de forma (aparentemente) paralela

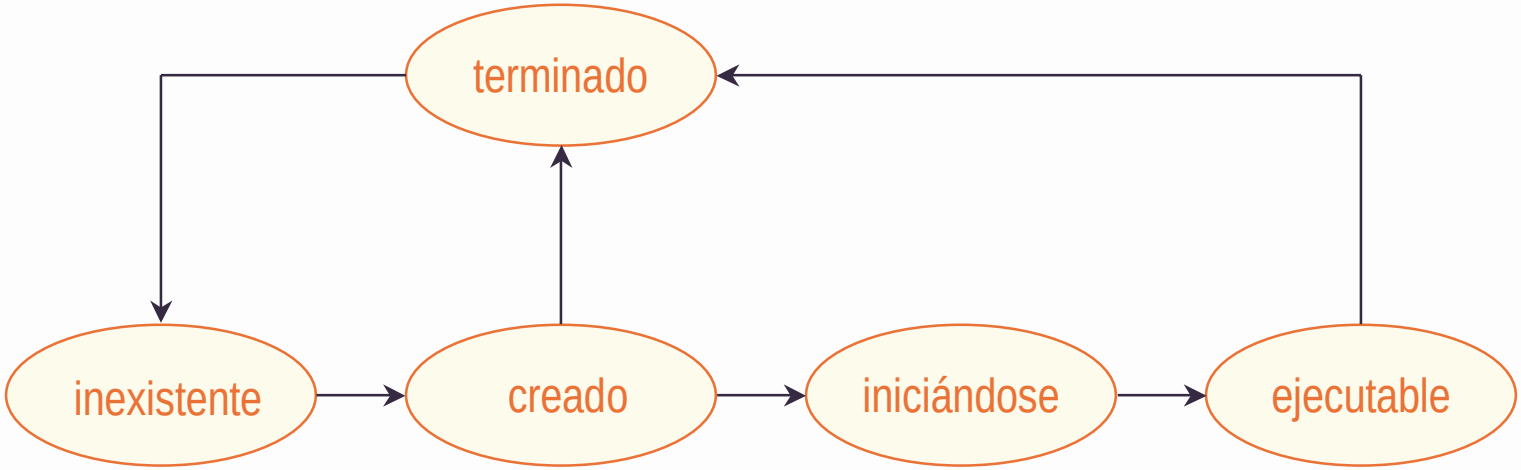
- Cada proceso tiene un flujo de control independiente
  - A veces se habla de procesos con varios flujos de control ***(threads)***
- Las instrucciones de los procesos se ejecutan intercalándose unas con otras (paralelismo lógico)

Índice

Ejecución concurrente

- La forma de ejecutar las tareas concurrentes puede variar, según se ejecuten:
  - En un sistema monoprocesador: (***multiprogramación***)
  - En un sistema multiprocesador(fuertemente acoplado) (***multiproceso***)
  - En un sistema distribuido(débilmente acoplado) (***procesamiento distribuido***)
- En todos los casos hay que multiplexar el uso del procesador o procesadores entre los procesos

Estados de un proceso



Núcleo de tiempo real

- Los procesos concurrentes se ejecutan con ayuda de un **núcleo de ejecución** *(run-time kernel)*
  - planificador *(scheduler)* en sistemas operativos
- El núcleo se encarga de la creación, terminación y el multiplexado de los procesos
- El núcleo puede tomar varias formas:
  - Núcleo desarrollado como parte de la aplicación
  - Núcleo incluido en el entorno de ejecución del lenguaje
  - Núcleo de un sistema operativo de tiempo real
  - Núcleo microprogramado
- El método de planificación que se usa afecta al comportamiento temporal del sistema

Procesos pesados y ligeros

- Los sistemas operativos suelen soportar procesos “pesados”
  - Cada proceso se ejecuta sobre una *máquina virtual* distinta
- Algunos sistemas operativos tienen procesos “ligeros” *(threads)*
  - Todos los *threads* de un mismo proceso comparten la misma máquina virtual
  - Tienen acceso al mismo espacio de memoria
  - El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias entre unos y otros
- La concurrencia puede estar soportada por el lenguaje o solo por el sistema operativo

Concurrencia en lenguajes de programación

Ventajas

- Programas más legibles y fáciles de mantener
- Programas más portátiles
- Se pueden detectar errores al compilar
- No hace falta sistema operativo

Inconvenientes

- Distintos modelos de concurrencia según lenguajes
- Puede ser difícil de realizar eficientemente
- Se puede mejorar la portabilidad con SO normalizados

Lenguajes concurrentes

- Tienen elementos de lenguaje para
  - Crear procesos concurrentes
  - Sincronizar su ejecución
  - Comunicarse entre sí
- Los procesos pueden
  - ser independientes,
  - cooperar para un fin común,
  - competir por el uso de recursos
- Los procesos que cooperan o compiten necesitan comunicarse y sincronizar sus actividades

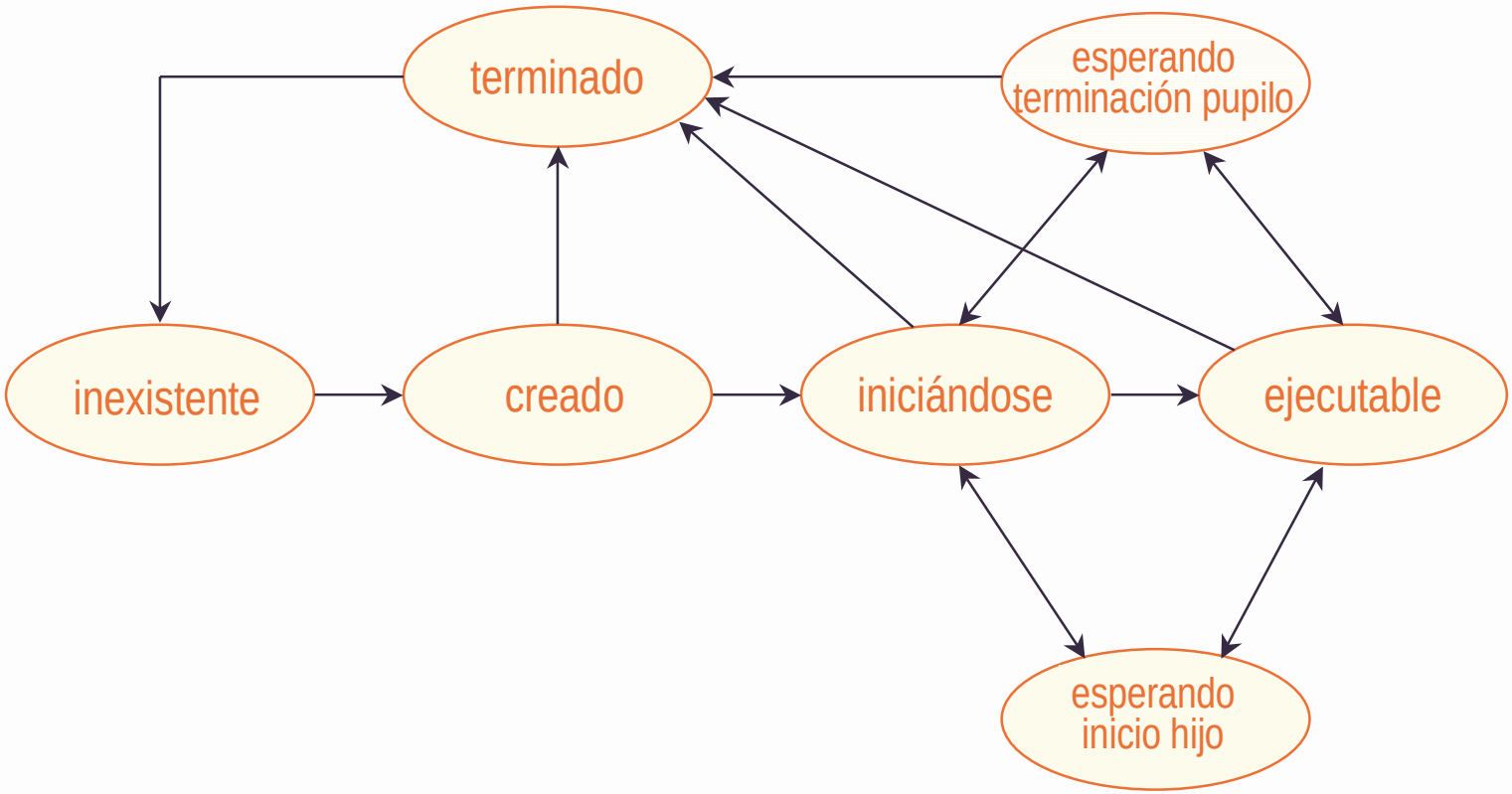
Características de los procesos

- **Estructura**
  - estática
  - **dinámica**
- **Nivel léxico**
  - plano
  - **anidado**
- **Granularidad**
  - **gruesa**
  - fina
- **Iniciación**
  - **paso de parámetros**
  - comunicación explícita
- **Terminación**
  - al completar la ejecución
  - “suicidio”
  - “aborto”
  - excepción sin manejar
  - cuando no hace falta
  - nunca

Jerarquía de procesos

- Proceso que crea otro
  - relación ***padre*** / ***hijo***
- Si el padre espera durante la creación e iniciación del hijo
  - relación ***tutor*** / ***pupilo***
  - el tutor puede ser
    - » el padre
    - » otro proceso
    - » un bloque interno de uno de los anteriores
  - el tutor no puede terminar hasta que lo hayan hecho todos sus pupilos

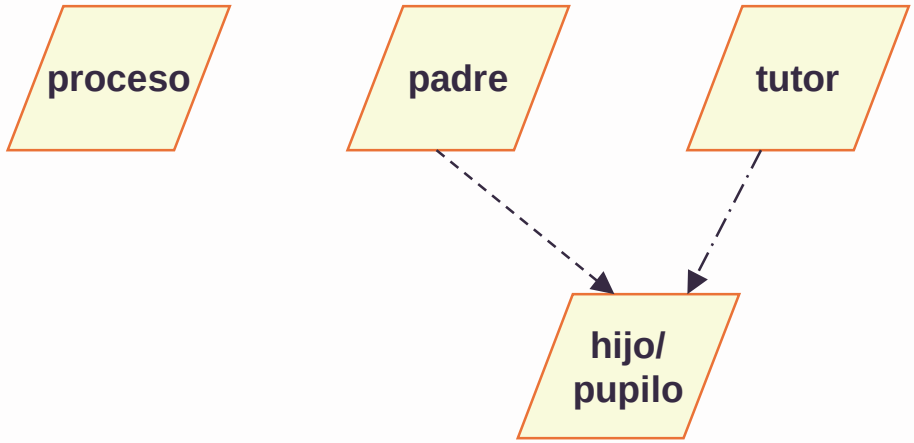
Estados de un proceso



Procesos y objetos

- Objetos **activos**
  - ejecutan acciones espontáneamente
  - tienen uno o más flujos de control *(threads)* propios
- Objetos **reactivos**
  - solo ejecutan acciones cuando se lo piden otros objetos
- Objetos **pasivos**
  - reactivos, sin control de acceso (se ejecutan sin restricciones)
- **Recursos**
  - reactivos, con control de acceso según su estado interno
  - necesitan un *agente de control*
    - » **Recursos protegidos**: agente pasivo
    - » **Servidores**: agente activo

Diagramas de procesos

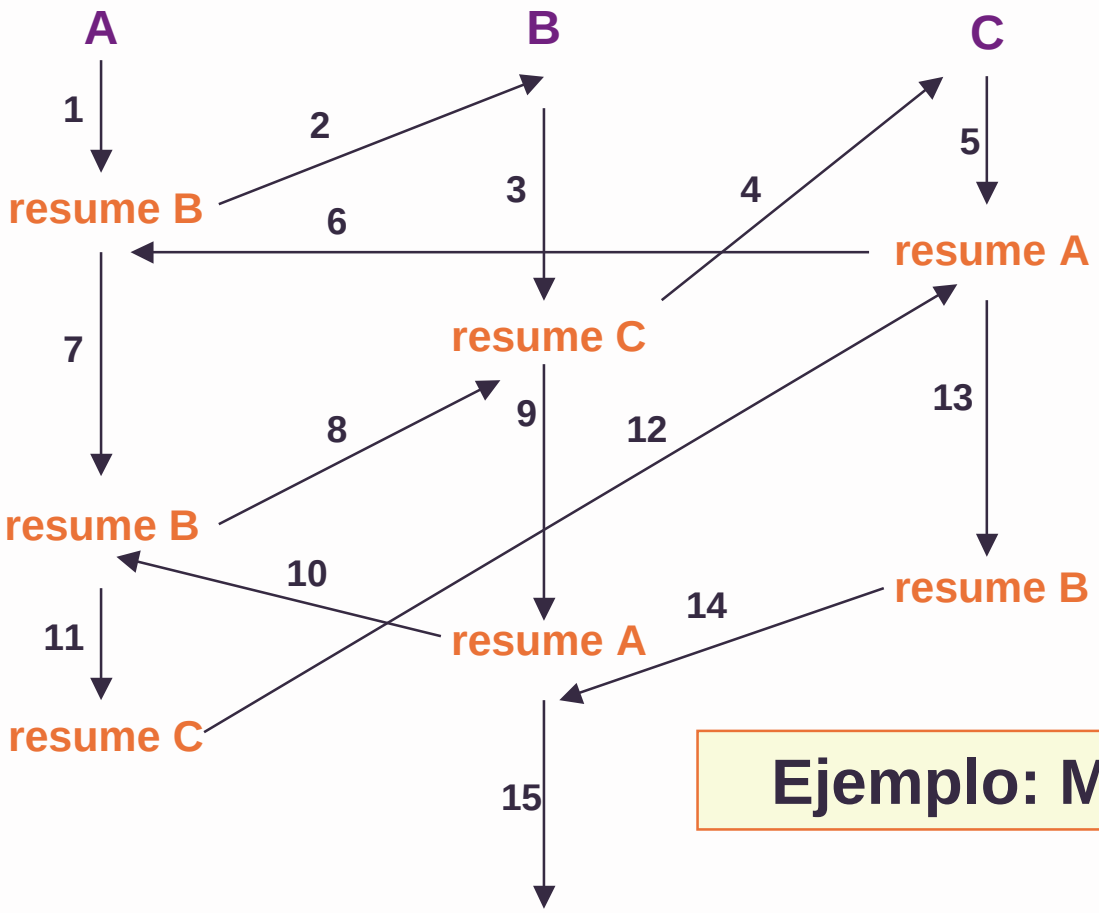


Índice

Representación de procesos

- Hay varias formas de representar la ejecución concurrente en un lenguaje de programación
  - Corrutinas
  - bifurcación y unión *(fork/join)*
  - *cobegin / coend*
  - declaración explícita de procesos
  - declaración implícita de procesos

Corrutinas



Ejemplo: Modula-2

fork / *join*

- Con *fork* se crea un nuevo proceso
- Con *join* se espera a que el proceso termine
- Ejemplo: C / POSIX.1

```
pid = fork(); /* se crea un proceso idéntico al original*/
...
c = join ();  /* el primero que llega espera al otro */
...
```

- Flexible, pero poco estructurado y propenso a errores
- En POSIX 1 se usa para crear *procesos pesados*

cobegin / coend

- Construcción estructurada

```
cobegin -- A, B y C se ejecutan concurrentemente
  A;
  B;
  C;
coend;  -- espera a que terminen A, B, y C
```

- Ejemplo: Edison, occam2

Declaración explícita de procesos

- Los procesos son unidades de programa (como los procedimientos)
- Esto permite una mejor estructura
- A veces hay también creación *implícita* de procesos
- Ejemplo: Ada, C/POSIX

Índice

Tareas en Ada

- En Ada, las actividades concurrentes reciben el nombre de **tareas**
- Las tareas tienen dos partes: *especificación* y *cuerpo*
  - En la especificación se define la interfaz visible de la tarea
    - » elementos de comunicación y otras cosas
  - En el cuerpo se describe la actividad de la tarea
- Las tareas se crean implícitamente cuando se entra se elabora su declaración
- Las tareas se empiezan a ejecutar antes que el cuerpo correspondiente a la parte declarativa donde se declaran

Ejemplo

```
procedure Example is
   task A; -- especificación
   task B;

   task body A is -- cuerpo
      -- declaraciones locales
   begin
      -- secuencia de instrucciones;
   end A;

   task body B is
      -- declaraciones locales
   begin
      -- secuencia de instrucciones;
   end B;

begin -- A y B empiezan a ejecutarse concurrentemente aquí
   -- secuencia de instrucciones de Example
   -- se ejecuta concurrentemente con A y B
end Example; -- no termina hasta que terminen A y B
```

Tareas y paquetes

```
package P is
   ...
end P;

package body P is

   task A;

   task body A is
      ...
   end A;

begin -- A empieza a ejecutarse aquí
   -- secuencia inicial de P
end P;
-- Si P es un paquete de biblioteca, la creación e iniciación de A
-- se ejecutan al elaborarse el paquete, dentro de la secuencia
-- inicial del programa.
-- El programa no termina hasta que haya terminado A
```

Tipos tarea

- A veces es útil tener un grupo de tareas similares
- La declaración de un tipo tarea permite disponer de una plantilla para crear tareas similares
- Las declaraciones de tareas individuales son semejantes a las de otros objetos
- Las tareas que se declaran directamente (como antes) son de un *tipo anónimo*

Ejemplo

```
task type A_Type;
task type B_Type;

A : A_Type;  -- se pueden declarar objetos en cualquier punto
B : B_Type;  -- donde sea visible la declaración

task body A_Type is
   ...
end A_Type;

task body B_Type is
   ...
end B_Type;
```

Estructuras

Se pueden declarar formaciones y registros con tareas

```
task type T;

A, B : T;

type Long is array (1 .. 100) of t;

type Mixture is
record
  Index  : Integer;
  Action : T;
end record;

task body T is
   ...
end T;
```

Discriminantes

```
procedure Main is
   type Dimension is (Xplane, Yplane, Zplane);
   task type Control (Dim : Dimension); -- Dim es un discriminante;
                  -- solo puede ser de un tipo discreto o de acceso
   C1 : Control (Xplane);
   C2 : Control (Yplane);
   C3 : Control (Zplane);

   task body Control is
      Position : Integer; -- posición absoluta
      Setting  : Integer; -- movimiento relativo
   begin
      Position := 0;
      loop
         New_Setting (Dim, Setting);
         Position := Position + Setting;
         Move_Arm (Dim, Position);
      end loop;
   end Control;

begin
   null;
end Main;
```

Tareas dinámicas

```
procedure Example is

   task type T;
   type A is access T;

   P : A;
   Q : A := new T; -- aquí se crea una nueva tarea
                   -- que se ejecuta inmediatamente;
                   -- la nueva tarea es Q.all
   task body T is ...

begin
   ...
   P := new T;  -- se crea otra tarea
   Q := new T;  -- y otra más
   -- la antigua Q.all sigue ejecutándose (pero es anónima)
   ...
end Example;
```

Master, amo o tutor

- El **tutor** (en Ada se llama *master* o *amo)* de una tarea dinámica es el bloque donde se declara el tipo acceso

```
declare
   task type T;
   type A is access T;
begin
   ...
   declare -- bloque interior
      X : T;           -- el amo de X es el bloque interior
      Y : A := new T; -- el amo de Y.all es el bloque exterior
   begin
      ...
   end;                -- espera a que termine X, pero no Y.all
   ...                 -- Y.all es anónima aquí
end;                   -- ahora espera a que termine Y.all
```

Identificación de tareas

- Los tipos acceso permiten dar un nombre a las tareas
- Los tipos tarea son privados y limitados

```
task type T;
type A is access T;

P : A := new T;
Q : A := new T;


...
P.all := Q.all; -- ilegal
P     := Q;     -- legal
```

- A veces se hace complicado
  - un mismo nombre puede designar tareas diferentes
- Es útil tener un *identificador* único para cada tarea

Ada.Task_Identification

```
package Ada.Task_Identification is

   -- definido en el anexo C - programación de sistemas

   type Task_Id is private;
   Null_Task_Id : constant Task_Id;

   function  "=" (Left, Right : Task_ID) return Boolean;

   function  Image (T : Task_ID) return String;

   function  Current_Task return Task_ID;

   procedure Abort_Task   (T : in out Task_ID);

   function  Is_Terminated(T : Task_ID) return Boolean;
   function  Is_Callable  (T : Task_ID) return Boolean;

private
   ...
end Ada.Task_Identification;
```

Además, *T'Identity* proporciona la identidad de cualquier tarea

Activación

- Consiste en la elaboración de las declaraciones locales de una tarea
  - Las tareas estáticas se activan justo antes de empezar la ejecución del padre
  - Las tareas dinámicas se activan al ejecutarse el operador **new**
- El padre espera a que termine la activación de las tareas que ha creado
- Si se produce una excepción durante la activación, se eleva *Tasking_Error* en el padre

Terminación

- Una tarea esta *completada* cuando termina la ejecución de su secuencia de instrucciones
- Una tarea completada *termina* cuando todos sus pupilos han terminado
- Una tarea puede terminar también cuando:
  - Se pone de acuerdo con otras tareas para terminar conjuntamente (lo veremos más adelante)
  - Es abortada por otra tarea o por sí misma
- El atributo booleano *T'Terminated* es verdadero cuando *T* ha terminado

Aborto

- Una tarea puede abortar otra tarea cuyo nombre es visible mediante la instrucción

```
abort T;
```

- Cuando se aborta una tarea, también se abortan todos sus pupilos
- Problema: no se puede abortar una tarea anónima
- Se puede abortar una tarea identificada mediante el procedimiento *Ada.Task_Identification.Abort_Task*

Índice

- Introducción
- Ejecución concurrente
- Representación de procesos
- Tareas en Ada
- ***Threads* en C / POSIX**
- Ejemplo

Procesos ligeros en C/POSIX.1c

```
/* pthread.h */

typedef ... pthread_t;       /* tipo thread */
typedef ... pthread_attr_t;  /* tipo atributos de thread */

int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);

int pthread_attr_setstacksize (...);
int pthread_attr_getstacksize (...);

int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg);
int pthread_join (pthread_t thread, void **value_ptr);
int pthread_exit (void *value_ptr);

pthread_t pthread_self (void);
...
```

Ejemplo (1)

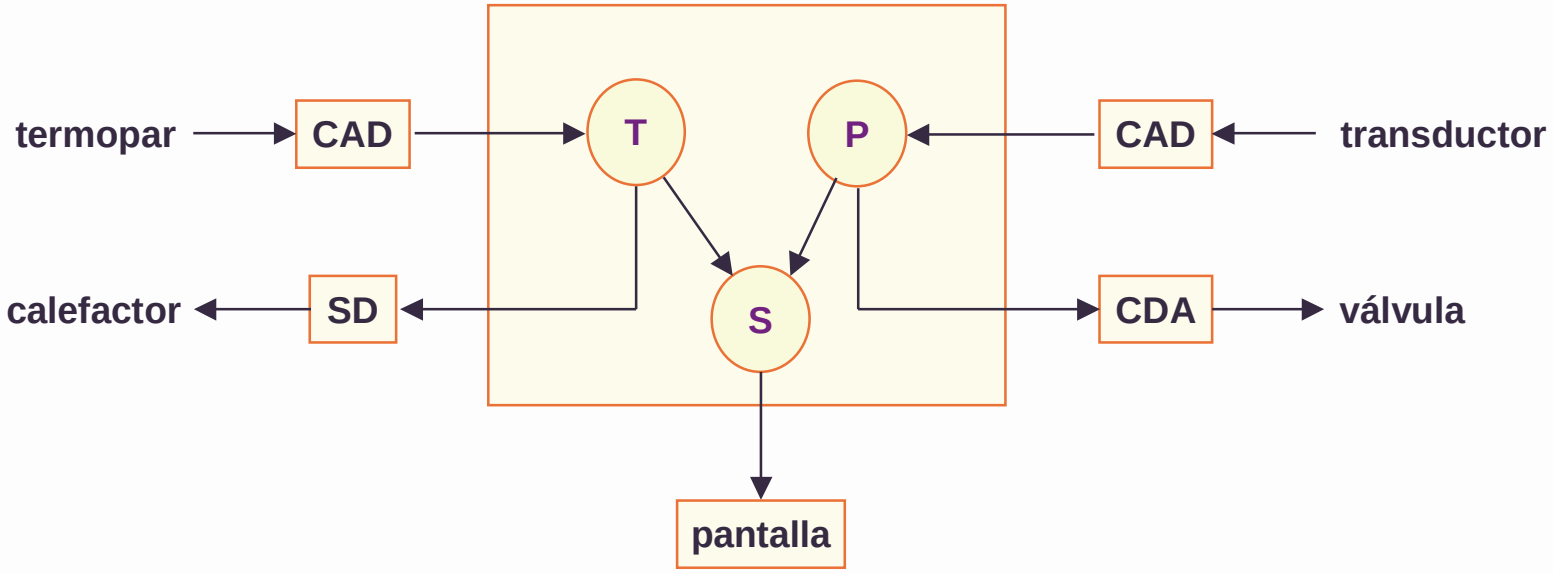
```
#include "pthread.h"

typedef enum {xplane, yplane, zplane} dimension;

int new_setting (int D);
void move_arm (int D, int P);

pthread_attr_t attributes;
pthread_t xp, yp, zp;

void *controller (dimension *dim) {
  int position, setting;

  position =0;
  while (1) {
     setting  = new_setting (dim);
     position = position + setting;
     move_arm (dim, position);
  };
  /* the thread executing controller does not terminate */
}
```

Ejemplo (2)

```
int main () {
   dimension X, Y, Z;
   void *result;

   X = xplane;
   Y = yplane;
   Z = zplane;

   if (pthread_attr_init(&attributes) != 0)
     exit (-1);
   if (pthread_create (&xp, &attributes,
                       (void *)controller, &X) != 0)
     exit (-1);
   if (pthread_create (&yp, &attributes,
                       (void *)controller, &Y) != 0)
     exit (-1);
   if (pthread_create (&zp, &attributes,
                       (void *)controller, &Z) != 0)
     exit (-1);

   pthread_join (xp, void **)&result); /* bloquear main */
}
```

Índice

Aplicación: sistema de control



Se trata de mantener la temperatura y la presión dentro de ciertos límites

Arquitectura de software

- Hay tres actividades concurrentes:
  - Control de presión.
  - Control de temperatura.
  - Visualización,
- Se puede hacer de tres formas:
  - Con un solo programa secuencial (**ejecutivo cíclico**)
    No hace falta sistema operativo
  - Con tres procedimientos secuenciales ejecutados como procesos de un **sistema operativo**
  - Con un solo **programa en un lenguaje concurrente**
    No hace falta sistema operativo, pero sí un núcleo de ejecución

Paquetes básicos

```
package Data_Types is
   type Temperature    is new Integer range 10 .. 500;
   type Pressure       is new Integer range  0 .. 750;
   type Heater_Setting is (On, Off);
   type Valve_Setting  is new Integer range 0..10;
end Data_Types;
```

```
package IO is
   procedure Read  (TR : out Temperature); -- CAD
   procedure Read  (PR : out Pressure);    -- CAD
   procedure Write (HS : Heater_Setting);  -- SD
   procedure Write (VS : Valve_Setting);   -- CDA
   procedure Write (TR : Temperature);     -- pantalla
   procedure Write (PR : Pressure);        -- pantalla
end IO;
```

```
with Data_Types; use Data_Types;
package Control_Procedures is
   procedure Control_Temperature (TR : Temperature;
                                  HS : out Heater_Setting);
   procedure Control_Pressure    (PR : Pressure;
                                  VS : out Valve_Setting);
end Control_Procedures;
```

Solución con ejecutivo cíclico

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  TR : Temperature;
  PR : Pressure;
  HS : Heater_Setting;
  VS : Valve_Setting;
begin
  loop
    Read (TR);
    Control_Temperature (TR, HS);
    Write (HS);
    Write (TR);
    Read  (PR);
    Control_Pressure (PR, VS);
    Write (VS);
    Write (PR);
  end loop;
end Controller;
```

Problemas

- La temperatura y la presión se muestrean con la misma frecuencia
  - se puede mejorar con un contador y estructuras *if*
- Mientras se espera leer una variable no se puede hacer nada con la otra
- Si un sensor falla, deja de leerse el otro también

Solución mejorada

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is
  ...
begin
  loop
    if Ready_Temperature then
      Read (TR);
      Control_Temperature (TR, HS);
      Write (HS);  Write (TR);
    end if;
    if Ready_Pressure then
      Read  (PR);
      Control_Pressure (PR, VS);
      Write (VS);  Write (PR);
    end if;
  end loop;
end Controller;
```

Crítica

- El programa pasa mucho tiempo en "espera ocupada"
  - es ineficiente
- Es difícil comprobar si funciona bien

El principal problema es que dos actividades independientes están acopladas entre sí

Concurrencia basada en el sistema operativo

```
package Operating_System_Interface is
  type Thread_Id   is private;
  type Thread_Code is access procedure;
  procedure Create_Thread (Code : Thread_Code; Id : out
Thread_Id);
  ...
private
  type Thread_Id is ...;
end Operating_System_Interface;
```

Solución con *threads* del sistema operativo (1)

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
with Operating_System_Interface; use Operating_System_Interface;
procedure Controller is
  Temperature_Controller, Pressure_Controller : Thread_Id;

  procedure Temperature_Control is
    TR : Temperature; HS : Heater_Setting;
  begin
    loop
    Read (TR); Control_Temperature (TR, HS); Write (HS); Write (TR);
    end loop;
  end  Temperature_Control;

  procedure Pressure_Control is
    PR : Pressure; VS : Valve_Setting;
  begin
    loop
      Read  (PR); Control_Pressure (PR, VS); Write (VS); Write (PR);
    end loop;
  end Pressure_Control;
```

Solución con *threads* del sistema operativo (2)

```
begin -- Controller
 Create_Thread (Temperature_Control'Access,
Temperature_Controller);
 Create_Thread (Pressure_Controler'Access,  Pressure_Controller);
 -- otras actividades iniciales
 -- esperar a que terminen Temperature_Controller y
Pressure_Controller
end Controller;
```

Crítica

- De esta forma se desacoplan completamente las dos actividades
- Con un sistema operativo real (p. ej. POSIX) sería mucho más complicado
- No es fácil ver qué procedimientos son concurrentes (código de *threads)*
- Mejor con soporte directo del lenguaje

Solución con un lenguaje concurrente (1)

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is

  task Temperature_Controller;
  task Pressure_Controller;

  task body Temperature_Controller is
    TR : Temperature; HS : Heater_Setting;
  begin
    loop
    Read (TR); Control_Temperature (TR, HS); Write (HS); Write (TR);
    end loop;
  end  Temperature_Controller;

  task body Pressure_Controller is
    PR : Pressure; VS : Valve_Setting;
  begin
    loop
      Read  (PR); Control_Pressure (PR, VS); Write (VS); Write (PR);
    end loop;
  end Pressure_Controller;
```

Solución con un lenguaje concurrente (2)

```
begin -- Controller
  null;
end Controller;
```

- Es más fácil de escribir y leer
- El código refleja directamente el paralelismo de la aplicación

Problemas

- Ambas tareas envían datos a la pantalla
  - hay que gestionar el acceso a este recurso compartido
  - si la escritura no se hace de forma exclusiva por cada tarea, el resultado es ilegible
- Hace falta un mecanismo para la gestión de recursos
  - sincronización de tipo exclusión mutua
  - hace falta el apoyo del entorno de ejecución

Resumen

- Los sistemas de tiempo real suelen realizar varias actividades en paralelo.
- Se pueden ejecutar estas actividades dentro de un programa secuencial mediante un ejecutivo cíclico, pero con serios inconvenientes.
- Los procesos concurrentes o tareas permiten descomponer un sistema en actividades concurrentes de forma natural.
- Las tareas de Ada son unidades de programa, con especificación y cuerpo.
- Las tareas de Ada también son objetos (limitados).
- Se pueden definir tipos tarea y tareas dinámicas con tipos de acceso.
- La norma POSIX.1c define una interfaz de sistema operativo para *threads* (procesos concurrentes con memoria compartida)