Sistemas en Tiempo Real

Grado en Ingeniería Electrónica

UNIDAD 6: Sincronización y comunicación basada en mensajes

Capítulo 6, excepto punto 6.9 de la edición Inglés (2009)

Capítulo 9, excepto punto 9.6 y apartado OCCA del punto 9.3 de la edición en Español (2003)

Comunicación y sincronización

El comportamiento correcto de un programa concurrente depende estrechamente de la sincronización y la comunicación entre tareas:

- **Sincronización**: satisfacer las restricciones en el entrelazado de las acciones de diferentes tareas.
- **Comunicación**: compartir información entre tareas.

Posibilidades:

- Uso de variables compartidas (Unidad 5)
- Paso de mensajes (Unidad 6)

Unidad 6: Paso de mensajes

La semántica de la comunicación basada en mensajes se define sobre tres cuestiones:

- El modelo de sincronización:
  - Asíncrono
  - Síncrono
  - Invocación remota
- El método de nombrado de tareas:
  - Directo/indirecto
  - Simetría/asimetría
- La estructura del mensaje

Sincronización de procesos

- **Asíncrono**: la tarea emisora no espera.
- **Síncrono**: la tarea emisora espera a que sea leído el mensaje.
- **Invocación remota**: la tarea emisora espera a la lectura del mensaje, la realización de sus tareas, y la generación de una contestación.

Eventos asíncronos:

```
P1
asyn_send (message)
```

```
P2
wait (message)
```

Eventos asíncronos para crear comunicación síncrona:

```
P1
asyn_send (message)
wait (acknowledgement)
```

```
P2
wait (message)
asyn_send (acknowledgement)
```

Eventos síncronos para crear invocación remota:

```
P1
syn_send (message)
wait (reply)
```

```
P2
wait (message)
  ...
  construct reply
  ...
syn_send (reply)
```

Nombrado de tareas

- Directo `send <message> to <task-name>`
- Indirecto `send <message> to <mailbox>`
  - Muchos a uno
  - Muchos a muchos
  - Uno a uno
  - Uno a muchos
- Simétrico

```
send <message> to <task-name>
wait <message> from <task-name>
```

```
send <message> to <mailbox>
wait <message> from <mailbox>
```

- Asimétrico `wait <message>`

Paso de mensajes en Ada

Definición de entradas para una tarea:

```
task Time_Server is -- a single task definition
  entry Read_Time(Now : out Time);
  entry Set_Time(New_Time : Time);
end Time_Server;
```

Definición de un tipo de tarea con entradas:

```
task type Screen_Output(Id : Screen_Identifier) is
  -- a task type definition
  entry Call (Value : Character; X_Coordinate,
              Y_Coordinate: Integer);
end Screen_Output;

Display: Screen_Output(Tty1);
-- where Tty1 is of type Screen_Identifier
```

Espera del mensaje (en el body de la tarea):

```
accept Call(C: Character; I,J : Integer) do
  Local_Array(I,J) : = C;
end Call;
```

Envío del mensaje desde otras tareas:

```
Time_Server.Read_Time(T);

Display.Call(Char,10,20);
```

Con tratamiento de errores:

```
begin
  Display.Call(C,I,J);
exception
  when Tasking_Error => -- log error
end;
```

Con comprobación previa de la tarea:

```
if Display'Terminated then
  -- log error and continue
```

Paso de mensajes en Ada

```
procedure Test is
  Number_Of_Exchanges : constant Integer := 1000;

  task T1 is
    entry Exchange (I : Integer; J : out Integer);
  end T1;

  task T2;

  task body T1 is
    A,B : Integer;
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- produce A
      accept Exchange (I : Integer; J : out Integer) do
        J := A;
        B := I;
      end Exchange;
      -- consume B
    end loop;
  end T1;

  task body T2 is
    C,D : Integer;
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- produce C
      T1.Exchange(C,D);
      -- consume D
    end loop;
  end T2;
begin
  null;
end Test;
```

Espera selectiva en Ada

- Sentencia **select**

```
task body Server is
  ...
begin
  loop
    -- prepare for service
    select
      accept S1(...) do
        -- code for this service
      end S1;
    or
      accept S2(...) do
        -- code for this service
      end S2;
    end select;
    -- do any housekeeping
  end loop;
end Server;
```

- Sentencia **select con guardia**

```
select
  when <Boolean-Expression> =>
    accept <entry> do
      ..
    end <entry>;
    -- any sequence of statements
or
  -- similar
  ...
end select;
```

Colas de mensajes en C/Real-Time POSIX

- En POSIX, la comunicación se realiza mediante colas de mensajes con primitivas **envía/recibe**.
- Soporta un esquema **simétrico asíncrono indirecto** y un mecanismo de comunicación **de muchos a muchos**.
- Una cola tiene atributos:
  - Tamaño máximo de la cola
  - Tamaño máximo de cada mensaje
  - Número de mensajes en cola
  - ……
- Al crear una cola se le da un nombre. Se accede a ellas de manera similar a los archivos

Colas de mensajes en C/Real-Time POSIX

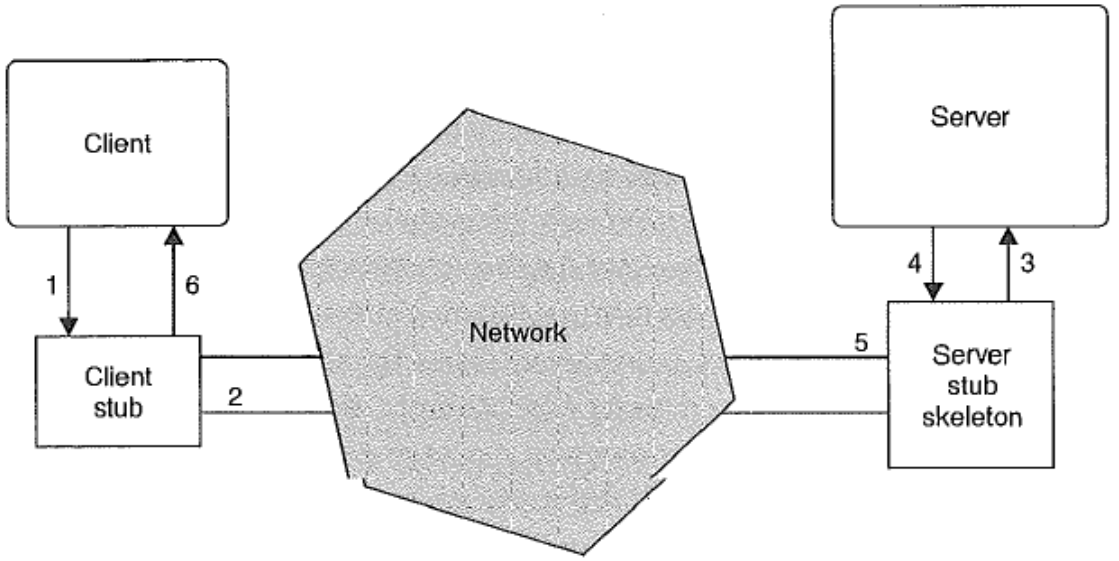
Funciones

- Manipulación de atributos: mq_getattr y mq_setattr
- Acceso a la cola: mq_open, mq_close
- Envío y recepción de mensajes: mq_send y mq_receive
  - Los datos son leídos/escritos de/hacia un búfer de caracteres. Si el búfer está vacío o lleno, el proceso emisor/receptor se bloquea, a menos que se haya activado el atributo O_NONBLOCK.
  - Si al desbloquearse una cola hay emisores y receptores esperando, no se indica cuál es despertado, a menos que se especifique la opción de planificación por **prioridad**. Si el proceso tiene múltiples hilos, cada hilo se considera un potencial emisor/receptor en sí mismo.
  - Una tarea puede indicar que se le envíe una señal cuando una cola vacía reciba mensajes y no haya receptores esperando. Esto permite implementar una espera selectiva.

Sistemas distribuidos

Hay 3 formas principales para la comunicación entre tareas:

- Una aplicación de interfaz externa (API)
- Paradigma Objetos distribuidos
- **Paradigma RPC: Remote Procedure Call**
  - Típicamente se usa para comunicaciones entre programas escritos en el mismo lenguaje.
  - Un procedimiento se identifica como servidor que puede ser llamado de manera remota.
  - Desde el servidor se pueden generar otros dos procedimientos (middleware) para permitir la comunicación de manera transparente:
    - client stub: se usa en lugar del servidor en el lado cliente
    - server stub: en lugar del servidor en el lado servidor



Remote Procedure Calls

- client stub:
  - Identificar la dirección del server stub
  - Marshalling de parámetros: transformarlos en un bloque de bytes para su transmisión
  - Solicitar la ejecución del proceso (request) al server stub
  - Esperar la respuesta del server stub y unmarshall los parámetros o excepciones
  - Devolver el control al cliente

- server stub:
  - Recibir peticiones del client stub
  - Unmarshalling de parámetros
  - Ejecutar la llamada al servidor
  - Recoger las posibles excepciones
  - Marshalling los parámetros o excepciones para su transmisión
  - Enviar la respuesta al client stub

Objetos distribuidos

Este paradigma o modelo permite:

- Crear dinámicamente un objeto en una máquina remota
- Identificar un objeto en cualquier máquina
- La invocación transparente de un método remoto en un objeto como si fuera local e independientemente del lenguaje en que esté escrito
- El envío transparente en tiempo de ejecución de una llamada a un método a través de la red.

En Ada:

Paquete Remote_Call_Interface.

- Un procedimiento puede recibir llamadas asíncronas (Asynchronous) indicándolo en su declaración.
- Partition_Id es un tipo que sirve para identificar particiones
- Params_Stream_Type se usa para el marshalling y unmarshalling
- Do_RPC y Do_APC (para asíncrono) envían el mensaje a la partición remota

En Java:

Paquete java.rmi. Contiene:

- la interfaz java.rmi.Remote.
- El paquete java.rmi.server:
  - RemoteServer (Clase **abstracta** que implementa Remote)
  - UnicastRemoteObject (clase **concreta** derivada de RemoteServer)

MÓDULO 1: CONCEPTOS BÁSICOS DE LOS SISTEMAS EN TIEMPO REAL

Este módulo explica los fundamentos de los sistemas de tiempo real y las características a tener en cuenta en su desarrollo. Se define el modelo de fallo y el concepto de tolerancia a fallos, indicando algunas técnicas para su resolución. Se termina el módulo explicando cómo se gestionan las excepciones (posibles errores) desde el punto de vista del lenguaje de programación y del propio sistema operativo (en tiempo real o no)

Autoevaluaciones de las unidades

- Auto-evaluación de la Unidad 1: Introducción a los sistemas en tiempo real
- Autoevaluación de la Unidad 2: Fiabilidad y tolerancia a fallos
- Autoevaluación de la Unidad 3: Excepciones y manejo de excepciones

MÓDULO 2: ASPECTOS DE LA CONCURRENCIA EN SISTEMAS EN TIEMPO REAL

Este módulo explica los conceptos de ejecución concurrente, que se basa en el modelo de procesos/tareas/hebras. Se indica cómo se implementa la concurrencia en los tres modelos de programación (Java RT, Ada y C/Real Time POFIX) para pasar a describir como se implementan los mecanismos de comunicación y sincronización mediante variables compartidas y mensajes. A continuación se introduce el concepto de acción atómica y la forma de implementarlas en los tres modelos de programación propuestos. Se finaliza el módulo explicando la gestión de acceso a los recursos y el modelo de interbloqueo

- Autoevaluación de la Unidad 4: Programación concurrente
- Autoevaluación de la Unidad 5: Sincronización y comunicación basada en variables compartidas
- Autoevaluación de la Unidad 6: Sincronización y comunicación basada en mensajes
- Autoevaluación de la Unidad 7: Acciones atómicas, procesos concurrentes y fiabilidad
- Autoevaluación de la Unidad 8: Control de recursos

Prueba de evaluación a distancia (PED1): Instalación/Configuración de un RTOS
Prueba de evaluación a distancia (PED2): Instalación/Configuración de RTSJ

MODULO 3: TEMPORIZACIÓN DE LAS TAREAS DE UN SISTEMA EN TIEMPO REAL

Este módulo explica los mecanismos de temporización que se pueden aplicar en un sistema de tiempo real, tomando como base el reloj del sistema en tiempo real. Una vez definidos los

Sistemas en Tiempo Real

Grado en Ingeniería Electrónica

UNIDAD 6: Sincronización y comunicación basada en mensajes

UNED

Nieves Pedreira Souto