Sistemas en tiempo real Documentation

Versión 0.1-rc

Roberto Castellor

06 de septiembre de 2017

1. Introducción a los Sistemas de Tiempo Real	1
1.1. Definición sistema en tiempo real	1
1.2. Tipos de sistemas en tiempo real	1
1.3. Ejemplos de sistemas en tiempo real	1
1.4. Características de sistemas en tiempo real	1
2. Fiabilidad y tolerancia a fallos	3
2.1. Fiabilidad, fallos y defectos	3
2.2. Modos de fallo	4
2.3. Estrategias para sistemas fiables	4
2.4. Nivel de tolerancia a fallos	5
3. Excepciones y manejo de excepciones	7
3.1. Propagación de excepciones	8
3.2. Modelos	8
4. Programación concurrente	9
4.1. Tipos de implementación	9
4.2. Mecanismos básicos de los lenguajes concurrentes	9
4.3. Comportamiento entre las tareas	10
4.4. Diferencias en la noción de tareas en los diferentes lenguajes	10
4.5. Las relaciones entre tareas	10
4.6. Mecanismos básicos para representar la ejecución concurrente	11
5. Sincronización y comunicación basada en variables compartidas	13
5.1. Espera ocupada	13
5.2. Suspende y reanuda	14
5.3. Semáforos	15
5.4. Regiones condicionales críticas	16
5.5. Monitores	16
6. Sincronización y comunicación basada en mensajes	17
7. Acciones atómicas, Tareas Concurrentes y Fiabilidad	19
7.1. Acciones atómicas	19
7.2. Acciones atómicas recuperables	20
7.3. Notificación asíncrona	20

7.4. Mecanismos de recuperación de errores	21
8. Control de recursos	23
8.1. Control de los recursos	23
8.2. Interbloqueo o bloqueo mutuo	23
9. Capacidades de tiempo real	25
10. Planificación	27
10.1. Modelo simple	27
10.2. Ejecutivo cíclico	28
10.3. Planificación basada en procesos	28
10.4. Alternativas de planificación	29
10.5. Protocolos de acotación de la prioridad	30
11. Programación de bajo nivel	33
12. Problemas de planificabilidad	35
12.1. Algoritmo de planificación de tasa monótonica	35
12.2. Test simple de planificabilidad para FPS	36
12.3. Estudio de planificación	36
13. Preguntas de exámenes	39
13.1. Mecanismos cambio de contexto	39
13.2. Fallos de temporización y valor	40
13.3. Primitivas de sincronización	40
13.4. Técnicas de detección de errores	41
13.5. Acciones atómicas	42
13.6. Manejo de excepciones	42
13.7. Interbloqueos	43
13.8. Protocolos de acotación de prioridad	43
13.9. Semántica del reencolado	45
13.10. Ejecución concurrente de procesos	45
13.11. Seguridad, fiabilidad y confiabilidad	45
13.12. Modelos abstractos de manejo de dispositivos	46
13.13. Modelo sincronización de procesos	47
13.14. Mecanismos de control de E/S	47
13.15. Interbloqueos	47
14. Indices y tablas	49

Introducción a los Sistemas de Tiempo Real

Definición sistema en tiempo real

Cualquier sistema en el que el tiempo en el que se produce la salida es significativo. Esto generalmente es porque la entrada corresponde a algún movimiento en el mundo físico, y la salida esta relacionada con dicho movimiento. El intervalo entre el tiempo de entrada y el de salida debe ser lo suficientemente pequeño para una temporalidad aceptable.

Cualquier actividad o sistema de proceso de información que tiene que responder a un estímulo de entrada generado externamente con un retardo finito y especificado.

La corrección de un sistema en tiempo real no depende sólo del resultado lógico de la computación, sino también del tiempo en el que se producen los resultados.

Tipos de sistemas en tiempo real

- **Hard (estrictos)** Son los sistemas en los que es absolutamente imperativo que las respuestas se produzcan dentro del tiempo límite especificado.
- **Soft (no estrictos)** Los tiempos de respuesta son importantes pero el sistema seguirá funcionando correctamente aunque los tiempos límite no se cumplan ocasionalmente.

Ejemplos de sistemas en tiempo real

Características de sistemas en tiempo real

Los sistemas en tiempo real tienen unas características específicas que los definen.

1. Funcionalidades en tiempo real
2. Control concurrente de sistemas separados

3. Programación de bajo nivel
4. Soporte a computación numérica
5. Grandes y complejos
6. Extremadamente fiables y seguros
7. Implementación eficiente y entorno de ejecución

Fiabilidad y tolerancia a fallos

Este tema, junto con el tema 3, trata de la producción de componentes de software fiables. Aunque se realizan consideraciones sobre la prevención de fallos, la atención se dedica principalmente a la tolerancia a fallos. Se condirean las técnicas de recuperación de errores hacia adelante y hacia atrás. El manejo de excepciones se estudia en el siguiente tema.

Causas que pueden propiciar el fallo de un sistema de tiempo real:

1. Especificación inadecuada.
2. Defectos provocados por errores de diseño.
3. Defectos provocados por fallos en componentes del procesador.
4. Defectos provocados por interferencias transitorias o permanentes en el subsistema de comunicaciones.

Los errores relacionados con el diseño o la especificación son difíciles de preveer mientras que los errores provocados por fallos son en cierto modo predecibles. Los lenguajes de programación de tiempo real tienen que ser altamente fiables.

Fiabilidad, fallos y defectos

Fiabilidad: una medida del éxito con el que el sistema se ajusta a alguna especificación definitiva de su comportamiento.

Fallo del sistema: Cuando el comportamiento de un sistema se desvía del especificado para él, se dice que es un fallo.

Se pueden distinguir tres tipos de fallos:

- Fallos transitorios, Un fallo transitorio comienza en un instante de tiempo concreto, se mantiene en el sistema durante algún periodo, y luego desaparece. Ejemplos de este tipo de fallos se dan en componentes hardware en los que se produce una reacción adversa a una interferencia externa, como la producida por un campo eléctrico o por radioactividad. Después de que la perturbación desaparece, lo hace también el fallo (aunque no necesariamente el error inducido). Muchos de los fallos en los sistemas de comunicación son transitorios.

- Fallos permanentes Los fallos permanentes comienzan en un instante determinado y permanecen en el sistema hasta que son reparados; es el caso, por ejemplo, de un cable roto o de un error de diseño de software.
- Fallos intermitentes Son fallos transitorios que ocurren de vez en cuando. Un ejemplo es un componente hardware sensible al calor, que funciona durante un rato, deja de funcionar, se enfría, y entonces comienza a funcionar de nuevo.

Modos de fallo

Se pueden identificar dos dominios generales de modos de fallo:

1. Fallos de valor, el valor asociado con el servicio es erróneo.
2. Fallos de tiempo, el servicio se completa a destiempo.

Las combinaciones de fallos de valor y de tiempo se denominan fallos arbitrarios.

Un fallo de valor fuera del rango esperado para el servicio se denomina error de límites, son fallos fácilmente reconocibles.

Los fallos en el dominio del tiempo se pueden englobar en:

1. Demasiado pronto, el servicio es entregado antes de lo requerido.
2. Demasiado tarde, el servicio se entrega después de lo requerido, se puede hablar de error de prestaciones.
3. Infinitamente tarde, el servicio nunca es entregado, fallo de omisión.

Esta clasificación se puede ampliar con fallos de encargo o improvisación cuando el servicio es entregado sin ser esperado.

Dada la clasificación de fallos se puede definir algunas suposiciones respecto al modo en que los sistemas pueden fallar:

1. Fallo descontrolado, un sistema que produce fallos arbitrarios tanto en el dominio del valor como del tiempo.
2. Fallo de retraso, un sistema produce servicios correctos en el dominio del valor pero no en el del tiempo.
3. Fallo de silencio, cuando el sistema falla bruscamente sin haber tenido fallos de valor o de tiempo, a partir del fallo todos los servicios subsiguientes también sufren fallos de omisión.
4. Fallo de parada, un sistema que cumple los requisitos de un fallo de silencio pero permite que otros sistemas detectan que ha entrado en el estado de fallo
5. Fallo controlado, un sistema falla de una forma especificada y controlada.
6. Sin fallos, un sistema produce los servicios correctos, tanto en el dominio del valor como del tiempo.

Estrategias para sistemas fiables

Para construir sistemas mas fiables podemos destacar la prevención de fallos limitando la introducción de fallos en la construcción del sistema con:

- Mejor hardware e interconexión de los componentes
- Aislar el hardware para la prevención de interferencias ambientales
- Especificación rigurosa del software con métodos de programación orientada a objetos
- Uso de herramientas especializadas del diseño del software

También podemos enfocarnos en la eliminación de fallos aunque a pesar de las pruebas realizadas para descubrir fallos nunca podemos demostrar su ausencia, las pruebas no pueden ser exhaustivas.

Nivel de tolerancia a fallos

El nivel de tolerancia a fallos lo podemos describir con este esquema:

- Tolerancia total: el sistema continua funcionando después de un fallo por un tiempo limitado, en este tiempo no se afecta ni la funcionalidad ni el rendimiento del sistema.
- Caída suave: el sistema sigue funcionando después de un fallo pero con aspectos de funcionalidad deteriorados
- Fallo seguro: se establece una parada segura que no produce efectos en la tolerancia respecto a otros componentes del sistema

Las técnicas de tolerancia a fallos se basan en la adición al sistema de elementos que permiten la recuperación, estos elementos se llaman redundantes en el sentido de que su uso no es necesario en caso del funcionamiento normal del sistema.

Respecto al hardware existen dos estrategias de tolerancia a fallos:

- Estática (enmascaramiento): módulos redundantes hardware que realizan la misma función, el sistema debe comparar los diferentes resultados para ser capaz de desechar los resultados incorrectos.
- Dinámica, la redundancia se produce dentro del propio componente, el mismo componente informa del error en la salida.

También podemos hablar de tolerancia a fallos en el software:

- Estático, Programación N-versiones, se trata de introducir diferentes componentes software que sean capaces de producir cálculos alternativos de la misma función y proporcionar resultados a comparar para asegurar que están libres de errores.

Los diferentes modelos deberían ser programados por personas, lenguajes y compiladores diferentes.

Los programas son ejecutados por un programa director que los ejecuta concurrentemente y serán comparados por el mismo. Los resultados (votos) deben ser idénticos, si alguno es diferente este resultado será considerado como erróneo.

Un problema que puede aparecer en los sistemas de medición de valores reales es la precisión de estos valores, necesitaremos entonces técnicas de comparación inexactas.

El éxito de la programación de N-versiones depende de:

- Correcta especificación inicial: la especificación del diseño provocará que los diferentes componentes software funcionen de forma correcta.
 - Independencia en el diseño: es importante que se utilicen diferentes personas y herramientas para la implementación de los diferentes componentes.
 - Presupuesto adecuado para desarrollar las tres versiones
- Redundancia dinámica

El error es informado por el propio módulo, que debe detectarlo. La detección puede ser por el entorno o por la aplicación.

El módulo debe ser capaz de confinar el error y valorar sus daños. El confinamiento consiste en establecer cortafuegos que delimiten el alcance del efecto de los errores. Se pueden realizar por:

- Descomposición modular: delimita la difusión del error en el módulo.
- Acciones atómica: en caso de error devuelve el sistema a un estado de consistencia

Recuperación de errores, hay que tratar de volver al estado de operación normal o de degradación en la funcionalidad. Existen dos tipos de recuperación:

- Recuperación hacia adelante, se continua con la ejecución del sistema haciendo correcciones selectivas de su estado
- Recuperación de errores hacia atrás, en caso de error el sistema se restaura a un estado seguro previo y se ejecuta desde ese punto - checkpoint

Una ventaja de la recuperación de errores hacia atrás es que el sistema no necesita localizar el error.

Como inconvenientes:

- No se puede recuperar de errores del entorno
- Efecto dominó, se retorna a puntos de recuperación debido a posibles datos incorrectos en la comunicación entre varios procesos.
- El error puede volver a ocurrir, debe ser eliminado en dos etapas: localización y reparación.

■ Bloques de recuperación

Se trata de bloques normales de programación con un punto de recuperación al principio y un test de aceptación al final que comprueba que el sistema se encuentra en un estado aceptable al finalizar la ejecución del bloque. Si existe un fallo se restaura el estado de ejecución al inicio del bloque continuando con un procedimiento alternativo. Si todas las alternativas presentan error el bloque mostrará un estado de error.

Es el test de aceptación el que se encarga de la detección de el error lo que puede suponer una sobrecarga. Por eso se introduce cierto margen de tolerancia para la aceptación.

Excepciones y manejo de excepciones

Se continúa con el estudio realizado en el tema anterior sobre la producción de componentes de software fiables, centrándose este tema en el manejo de las excepciones, estudiándose tanto los modelos de terminación como los de reanudación.

Los mecanismos de manejo de excepciones tienen que cumplir estos requisitos:

1. El mecanismo debe ser fácil de comprender y utilizar.
2. El código de manejo de excepciones no debe oscurecer el código normal del programa.
3. El mecanismo debe diseñarse de forma que no suponga una sobrecarga en la ejecución. Solamente será aceptable cierta pequeña sobrecarga cuando sea primordial la rapidez de recuperación al ocurrir una excepción.
4. El mecanismo debe permitir un tratamiento uniforme de las excepciones tanto por el entorno como por el programa.
5. El mecanismo debe permitir la programación de acciones de recuperación.

Existen métodos de control de excepciones en lenguajes tipo C implementados con valores de retorno inusuales, normalmente distintos de cero. Otro método de control de excepciones en este tipo de lenguajes es mediante la bifurcación forzada, por ejemplo el uso de `goto` en C aunque este tipo de control de excepciones generan un código más oscuro.

Actualmente en los lenguajes de programación modernos se tiende a incluir funcionalidades de manejo de excepciones directamente en el lenguaje, lo que provee un mecanismo más estructurado.

Según el retraso en la detección del error las excepciones se provocan de forma síncrona o asíncrona, la excepción síncrona se genera como respuesta a un bloque de código, la excepción asíncrona se genera un tiempo después de que ocurra la operación que da lugar a la aparición del error.

Existen cuatro tipos de excepciones:

1. Detectada por el entorno y generada síncronamente, por ejemplo una división por cero.
2. Detectada por la aplicación y generada síncronamente, por ejemplo un fallo en un aserto.
3. Detectada por el entorno y generada asíncronamente, por ejemplo un fallo de alimentación o un mecanismo de monitorización vital.

4. Detectada por la aplicación y generada asíncronamente, por ejemplo cuando un proceso verifica una condición que ocasionará que un proceso no cumpla los plazos o no termine correctamente.

Existen diferentes formas de declarar las excepciones síncronas:

1. Mediante un nombre constante que necesita ser declarado explícitamente.
2. Mediante un objeto de cierto tipo que podrá o no ser declarado explícitamente.

Ada precisa que las excepciones se declaren como constantes, por ejemplo las excepciones que puede generar el entorno de ejecución se declaran en el paquete Standard.

```
package Standard is
...
Constraint_Error : exception;
Program_Error : exception;
Storage_Error : exception;
Tasking_Error : exception;
...
end Standard
```

Java y C++ dan una visión de las excepciones más orientada al objeto. En java las excepciones son instancias de una subclase de la clase Throwable y podran ser lanzadas tanto por el sistema de ejecución como por la aplicación. En C++ se pueden lanzar excepciones de cualquier tipo de objeto sin declararlas previamente.

Se considera que un bloque esta vigilado (guarded) cuando se ha definido un manejador de excepciones para el, por ejemplo en Java se considerará vigilado el bloque en una estructura try ... catch. Al definir los bloques podemos incurrir en un problema de granularidad inadecuada cuando el bloque es demasiado grande y no podemos definir en que instrucciones se produce la excepción. Una solución a este problema es disminuir el tamaño del bloque y/o anidarlos. Otra alternativa es crear procedimientos con manejadores para cada uno de los bloques anidados. La mejor aproximación es permitir el paso de parametros junto a las excepciones.

Propagación de excepciones

Cuando no hay un manejador de excepciones en un bloque de código y se produce una excepción hay dos formas de proceder:

- Entender la ausencia de manejador como un error de programación y notificarlo en tiempo de compilación.
- La segunda vía es buscar manejadores en el pasado de la cadena de invocación en tiempo de ejecución, esto se conoce como propagación de excepciones.

Modelos

Existen varios modelos de excepción:

- Modelo de continuación, en el caso de que el invocador sea capaz de manejar la excepción propagada por el proceso invocado, este aplicará las acciones necesarias y continuará con su ejecución.
- Modelo de terminación, cuando el control de la excepción no se devuelve al invocador no existen acciones de manejo que termina la ejecución del proceso.
- Modelos híbridos, el manejador es el que decide si continuar con la ejecución del proceso o terminar la ejecución

Programación concurrente

En este tema se introduce la noción de proceso, tarea y hebra o hilo y revisa los modelos que utilizan los diseñadores de lenguajes y de sistemas operativos. El término tarea se utiliza genéricamente para representar una actividad concurrente. Se aprovecha también este tema para estudiar la distribución de tareas cuando se dispone de multiprocesadores o de sistemas distribuidos. Dejándose para los siguientes dos temas la comunicación entre tareas.

La programación concurrente es el nombre que se le da a las notaciones y expresiones que permiten expresar paralelismo potencial y resolver problemas de sincronización y comunicación.

- Un programa secuencia -> contiene un único camino de ejecución.
- Un programa concurrente son varios procesos secuenciales autónomos y en paralelismo potencial.

En los sistemas operativos un proceso contiene un hilo de control que normalmente se ejecuta en su propia máquina virtual. Se crean diferentes hilos de proceso dentro de un programa que tienen acceso libre a la memoria compartida por este, estos hilos se denominan **hebras o tareas**.

Las hebras o tareas que se ejecutan independientemente del sistema operativo y a nivel de aplicación se llaman **fibras**.

Tipos de implementación

- Se multiplexa la ejecución de las diferentes tareas en un único procesador **multiprogramación**
- Se ejecutan las tareas en un sistema multiprocesador con memoria compartida **multiprocesamiento**
- Se ejecutan en varios procesadores sin memoria compartida **Sistemas distribuidos**

El RTTS (sistema de soporte de tiempo de ejecución) se sitúa entre el hardware y la aplicación para gestionar las diferentes tareas.

Mecanismos básicos de los lenguajes concurrentes

- Expresión de actividades concurrentes

- Sincronización entre las actividades concurrentes.
- Primitivas de soporte de comunicación entre tareas.

Comportamiento entre las tareas

- Independientes -> No se comunican ni se sincronizan en ningún momento.
- Cooperativas -> necesitan comunicarse y sincronizarse entre ellas.
- Competitivas -> se dan casos en los que se dispone de recursos limitados, las tareas compiten para usar estos recursos.

Diferencias en la noción de tareas en los diferentes lenguajes

- Estructura
 - Estática -> el número de tareas no varía y se conoce en tiempo de compilación.
 - Dinámica -> varía el número de tareas en tiempo de ejecución.
- Nivel
 - Anidado -> las tareas se definen en cualquier nivel.
 - Plano -> todas las tareas son definidas en el mismo nivel
- Granularidad
 - Gruesa -> pocas tareas con larga duración.
 - Fina -> muchas tareas con poca duración.
- Inicialización y terminación

Las relaciones entre tareas

Las relaciones entre tareas pueden dar paso a una **jerarquía de tareas**:

- Relación padre/hijo.
- Relación guardián/dependiente -> El guardián no puede terminar su ejecución hasta que finalicen todos sus dependientes.

En el paradigma orientado a objetos los objetos puede ser:

- Activos -> ejecutan acciones por si mismos
- Reactivos -> solo realizan acciones al ser invocados por un objeto activo

Todo esto da lugar a diferentes tipos de entidades:

- Pasiva -> reactiva sin restricciones de sincronización
- Recurso protegido -> reactivo con restricciones, el acceso es por tanto protegido.

Los objetos activos tienen un hilo interno de ejecución.

Mecanismos básicos para representar la ejecución concurrente

- Fork/join, especifica que la rutina invocada debe ejecutarse concurrentemente con el invocador
- Cobegin, es una estructura que permite la ejecución concurrente de una serie de sentencias que se especifican como sigue: cobegin S1: S2: La ejecución termina cuando terminan de ejecutarse todas las sentencias.
- Declaración explícita de tareas, se ha convertido en el mecanismo por excelencia para los lenguajes con mecanismos de programación en sistemas de tiempo real. Existen dos mecanismos principales para la declaración explícita de tareas en java:
 - Creación de hilos extendiendo a la clase **java.lang.Thread**.
 - Implementación de la interfaz “**Runnable**” cuyo método run permite la ejecución concurrente al ser sobrescrito.

Sincronización y comunicación basada en variables compartidas

Este tema, junto con el siguiente, se centra en el estudio de la comunicación entre tareas. En concreto en este primer tema se escriben los métodos de variables compartidas, incluyendo la utilización de semáforos, monitores, variables compartidas y objetos protegidos.

El comportamiento correcto de un programa concurrente depende estrechamente de la sincronización y la comunicación entre procesos. Sincronizar es satisfacer las restricciones en el entrelazado de las acciones de diferentes procesos. Algunas formas de comunicación requieren sincronización y la sincronización puede ser considerada como comunicación sin contenido.

La comunicación entre procesos se basa normalmente o en el uso de variables compartidas o en el paso de mensajes. Las variables compartidas son objetos a los que puede acceder más de un proceso, la comunicación puede realizarse referenciando en cada proceso dichas variables cuando sea apropiado. El paso de mensajes implica el intercambio explícito de datos entre dos procesos mediante un mensaje que pasa de un proceso a otro siguiendo algún mecanismo. Hay que señalar que la elección entre variables compartidas y paso de mensajes deben realizarla los diseñadores de los lenguajes o de los sistemas operativos y no implica que deba utilizarse un método particular de implementación. Las variables compartidas son más fáciles de soportar si hay memoria compartida entre procesos, pero pueden ser utilizadas incluso si el hardware incorpora un medio de comunicación. Este tema se centrará en las primitivas de sincronización y comunicación basadas en memoria compartida. En particular se verán los conceptos de espera ocupada, semáforos, regiones críticas condicionales, monitores, tipos protegidos y métodos sincronizados.

Una secuencia de sentencias que deben aparecer como ejecutada indivisiblemente se denomina sección crítica. La sincronización que se precisa para proteger una sección crítica se conoce como exclusión mutua. La exclusión mutua no es la única sincronización importante. La sincronización condicionada, o de condición, es otro requisito significativo, y es necesaria cuando un proceso desea realizar una operación que sólo puede ser realizada adecuadamente, o de forma segura, si otro proceso ha realizado alguna acción o está en algún estado definido.

Espera ocupada

Una forma de implementar la sincronización es comprobar las variables compartidas que actúan como indicadores en un conjunto de procesos. Esta aproximación sirve razonablemente bien para implementar sincronización de condición, pero no hay un método simple para la exclusión mutua. Para indicar una condición, un proceso activa el valor de un

indicador; para esperar por esta condición, otro proceso comprueba este indicador y sólo continúa cuando se lee el valor apropiado:

```
process P1; // proceso esperando
...
while indicador = abajo do
    null
end;
...
end P1;
process P2; // proceso indicando
...
indicador := arriba;
...
end P2;
```

Si la condición no es aún correcta, P1 no tiene elección y deberá continuar en el bucle para volver a comprobar el indicador. Esto se conoce como *espera ocupada*, y también como *giro -y* a los indicadores como *cerrojos de giro* (spin locks)-.

Los algoritmos de espera presentan algunas dificultades que pueden resumirse como:

- Los protocolos de espera ocupada son difíciles de diseñar y comprender, y es complicado probar su corrección.
- Los programas de prueba pueden ignorar entrelazamientos raros que rompen la exclusión mutua o llevan a un interbloqueo activo.
- Los bucles de espera ocupada son ineficientes.
- Una tarea no fiable que utilice falsamente las variables compartidas, corromperá el sistema completo.

Ningún lenguaje de programación concurrente se basa completamente en espera ocupada y variables compartidas, hay otros métodos y primitivas, los semáforos y monitores se describen en las siguientes secciones.

Suspender y reanudar

Un método alternativo a la espera ocupada es suspender (eliminar del conjunto de procesos ejecutables) al proceso solicitante si la condición por la que espera no es cierta. Por ejemplo

```
process P1; //proceso que espera
...
if indicador = abajo do
    suspend;
end;
indicador := abajo;
...
end P1;
process P2; // signaling process
...
indicador := arriba;
resume P1; // no tiene efecto si P1 no ha sido suspendido
...
end P2;
```

El ejemplo anterior se puede representar en Java así:

```
boolean indicador;
final boolean arriba = true;
```

```

final boolean abajo = false;

class PrimerHilo extends Thread {
    public void run() {
        ...
        if(indicador == abajo) {
            suspend();
        }
        indicador = abajo;
        ...
    }
}

class SegundoHilo extends Thread {
    PrimerHilo H1;

    public SegundoHilo(PrimerHilo H){
        super();
        H1 = H;
    }

    public void run() {
        ...
        indicador = arriba;
        H1.resume();
        ...
    }
}

```

Esta aproximación padece lo que se conoce como una condición de carrera. El hilo H1 podría comprobar el indicador y el soporte de ejecución decidir desalojarlo y ejecutar H2. H2 activa el indicador y resume H1. H1 no está suspendido por lo que resume no tiene efecto. Ahora cuando H1 vuelva a ejecutarse, piensa que indicador está abajo y se suspende el mismo.

La razón de este problema es que el indicador es un recurso compartido que está siendo comprobado, y una acción que va a ser tomada depende de sus estatus. Esta comprobación y suspensión no es una acción atómica y por tanto puede ser interferenciada por otros procesos. Java da por obsoletos estos métodos.

Para resolver los problemas de condición de carrera se opta por realizar una *suspensión en dos etapas*. P1 debe anunciar que plantea suspenderse próximamente, cualquier operación de reanudación que encuentre con que P1 no está suspendido verá aplazado su efecto y cuando P1 se suspenda, será reiniciado inmediatamente, es decir, anulará la suspensión.

Semáforos

Los semáforos son un mecanismo sencillo para la programación de la exclusión mutua y la sincronización de condición. Diseñados originalmente por Dijkstra presentan los siguientes beneficios:

- Simplifican los protocolos para la sincronización.
- Eliminan la necesidad de bucles de espera ocupados.

Un *semáforo* es una variable entera no negativa que, aparte de la inicialización, sólo puede ser manejada por dos procedimientos. Estos procedimientos fueron llamados por Dijkstra P y V, pero en estos apuntes los denominaremos *wait* y *signal*. La semántica de wait y signal es como sigue:

- wait(S) Si el valor del semáforo, S, es mayor que cero, entonces decrementa su valor en uno; en caso contrario, demora el proceso hasta que S sea mayor que cero (y entonces decrementa su valor).
- signal(S) Incrementa el valor del semáforo, S, en uno.

Estas operaciones son atómicas.

Regiones condicionales críticas

En estas regiones se agrupan las variables protegidas y sentencias indivisibles. El acceso a la región condicional crítica esta protegido por una variable booleana denominada guarda.

Monitores

Los monitores aparecen como mecanismos que intentan solucionar estos problemas. En ellos las regiones críticas se escriben como procedimientos y por tanto las variables protegidas están ocultas.

El programador no se debe preocupar por la exclusión mutua en la llamada de procedimientos de monitor ya que estas están serializadas.

Existe un tipo de monitores que garantiza la indivisibilidad de las tareas internas de estos, el llamado **monitor de Hoore**. En el existen dos señales: wait y signal, la primera bloquea el proceso que la llama y la segunda libera un proceso de la cola de procesos bloqueados, sino existe ningún proceso en esta cola la llamada signal no tiene efecto.

Sincronización y comunicación basada en mensajes

Este tema es la continuación del anterior y resalta la importancia que tienen en los lenguajes modernos los métodos basados en mensajes para la comunicación y sincronización.

La sincronización basada en mensajes se basa en tres fases y/o clasificaciones:

- Modelo de sincronización: las variaciones en el modelo de sincronización dependen de la semántica de envío.
 - Envío asíncrono, el emisor envía un mensaje y continúa inmediatamente con sus acciones después de enviarlo. Requiere el uso de buffers.
 - Envío síncrono, el emisor envía el mensaje y no continúa hasta que recibe el acuse de recibo del receptor.
 - Invocación remota, el emisor espera a la elaboración de la respuesta por parte del receptor. La espera es más larga que en el envío síncrono.
- Nombrado de procesos:
 - Nombrado directo: se hace referencia explícita al receptor
 - Nombrado indirecto: se usa una entidad intermedia para el envío (canal, tubería, etc...)
 - Nombrado simétrico: el emisor y el receptor se envían en mensajes entre sí. Especificado en todo momento los nombres.
 - Nombrado asimétrico, no se nombra la fuente específica de recepción (Cliente/Servidor)
- Estructura de mensajes: algunos lenguajes de programación concurrentes restringen la estructura de sus mensajes.

En la sincronización los procesos deben estar preparados para la comunicación, si uno está preparado el otro deberá esperar a que lo esté.

Espera selectiva, el receptor(servidor) tiene que esperar a que quede libre el canal de comunicación. Podemos permitir esperar a varios procesos realizando una selección en el servidor, mediante el uso de guardas se elige el procedimiento que se ejecuta en el servidor.

Se puede dar el caso de que varias guardas se evalúen como verdadera, en este caso la ejecución será indeterminista, es decir, se elegirá la opción aleatoriamente.

Se produce una cita cuando un cliente invoca a un servicio de un servidor y este acepta.

Acciones atómicas, Tareas Concurrentes y Fiabilidad

En este tema se amplian las discusiones iniciales sobre tolerancia a fallos describiendo con cuánta fiabilidad puede ser programada la cooperación entre procesos. Para esta discusión es fundamental la noción de acción atómica y las técnicas de manejo de eventos asíncronos.

Los objetivos del tema pasan por:

Acciones atómicas

Una acción es atómica si las tareas que la llevan a cabo no son conscientes de las actividades de las primeras, durante el tiempo en el que llevan a cabo la acción atómica.

- Sin comunicación con otras tareas
- No detectan cambios de estado externos
- Los cambios de estado internos no se comunican hasta finalizar

Las acciones atómicas son vistas desde fuera como indivisibles e instantáneas.

Para permitir la descomposición modular de las acciones atómicas se incorpora la noción de acciones atómicas anidadas. Los procesos comprometidos en una acción anidada deben ser un subconjunto de los que están involucrados en el nivel externo de la acción.

Las propiedades que tiene una acción atómica:

- Una acción es atómica si los procesos que la realizan no saben de la existencia de ningún otro proceso activo, y ningún otro proceso activo tiene constancia de las actividades de los procesos durante el tiempo que en el que están realizando la acción.
- Una acción es atómica si los procesos que la realizan no se comunican con otros procesos mientras está siendo realizada la acción.
- Una acción es atómica si los procesos que la realizan no pueden detectar ningún cambio de estado salvo aquellos realizados por ellos mismos, y si no revelan sus cambios de estado hasta que la acción se haya completado.

- Las acciones son atómicas si, en lo que respecta a otros procesos, pueden ser consideradas indivisibles e instantáneas, de forma que los efectos sobre el sistema sean como si estuvieran entrelazadas y no en concurrencia.

Acciones atómicas recuperables

La expresión transacciones atómicas se ha utilizado frecuentemente en el marco conceptual de los sistemas operativos y las bases de datos. Una transacción atómica tiene todas las propiedades de una acción atómica, más la característica adicional de que su ejecución puede tener éxito o fallar. Por fallo se entiende la ocurrencia de un error del que la transacción no puede recuperarse, por ejemplo, un fallo de procesador. Si falla una unidad atómica, los componentes del sistema que están siendo manipulados por la acción pueden quedar en estado inconsistente. Ante un fallo, una transacción atómica garantiza que los componentes son devueltos a su estado original. Las transacciones atómicas a veces se conocen como acciones recuperables.

Las dos propiedades distintivas de las transacciones atómicas son:

- Atomicidad de fallo, lo que significa que la transacción debe o bien ser completada con éxito o (en el caso de fallar) no tener efecto.
- Atomicidad de sincronización (o aislamiento), lo que significa que la transacción es indivisible, en el sentido de que su ejecución parcial no puede ser observada por ninguna transacción que se esté ejecutando concurrentemente.

Notificación asíncrona

El requisito fundamental de un mecanismo de notificación asíncrona es permitir que un proceso responda rápidamente a una condición que ha sido detectada por otro proceso. El énfasis está en la prontitud de la respuesta, obviamente, un proceso siempre puede responder a un evento simplemente sondeando o esperando ese evento. La notificación del evento puede construirse sobre el mecanismo de comunicación y sincronización de proceso. El proceso, cuando está disponible para recibir el evento, simplemente realiza la petición apropiada.

Desafortunadamente, hay ocasiones en las que no es adecuado esperar o sondear la ocurrencia de eventos, como en las siguientes:

- *Recuperación de errores* Cuando hay grupos de procesos que realizan acciones atómicas, la detección de un error en un proceso necesita de la participación del resto de procesos en la recuperación. Por ejemplo, un fallo hardware puede suponer que el proceso nunca termine su ejecución prevista porque las condiciones de comienzo ya no se cumplen; el proceso puede que nunca alcance su punto de sondeo. También podría ocurrir un fallo de temporización, por lo que no se podría cumplir el plazo límite de su servicio. En estas situaciones el proceso debe ser informado de que ha sido detectado un error y de que debe efectuar alguna forma de recuperación de error lo más rápidamente posible.
- *Cambios de modo* Los sistemas de tiempo real normalmente tienen distintos modos de operación. En ocasiones habrá cambios de modo, y estos deben ser gestionados en puntos bien definidos de la ejecución del sistema. Sin embargo, en algunas áreas de aplicación, los cambios de modo pueden ser esperados pero no planificados. En estas situaciones, los procesos deben ser informados de forma rápida y segura de que el modo en el que estaban operando ha cambiado y deben proceder a un conjunto de acciones diferentes.
- *Planificación utilizando computaciones parciales/imprecisas* Existen muchos algoritmos en los que la precisión de los resultados depende del tiempo asignado a los cálculos. Por ejemplo, los cálculos numéricos, computaciones estadísticas y búsquedas heurísticas, generan una estimación inicial del resultado requerido, que después es mejorada con una mayor precisión. En tiempo de ejecución, a cada algoritmo se le asigna una cierta cantidad de tiempo, transcurrido el cual el proceso debe interrumpirse, de forma que se finaliza el mecanismo de refinamiento del resultado.

- *Interrupciones de usuario* En un entorno interactivo general, los usuarios a menudo desean finalizar el procesamiento actual porque han detectado una condición de error y desean comenzar de nuevo.

Una aproximación al manejo asíncrono de notificación es abortar el proceso y permitir que otro proceso efectúe alguna forma de recuperación. Todos los sistemas operativos y la mayoría de los lenguajes de programación concurrente proporcionan esta disponibilidad. Sin embargo, el aborto de un proceso puede ser costoso y a menudo es una respuesta extrema a muchas condiciones de error. Consecuentemente, es necesario algún mecanismo de notificación asíncrona.

Mecanismos de recuperación de errores

Mecanismos de recuperación de errores en las acciones atómicas:

- Recuperación de errores hacia atrás: Si existe un error en la acción atómica todas las tareas involucradas retroceden. El mecanismo que se encarga de esta recuperación se llama **conversación**
 - Se guarda el contexto de los procesos al inicio de la conversación.
 - La comunicación de los procesos es solo con otros procesos internos o con el gestor de recursos.
 - Para abandonar la conversación todos los procesos deben haber pasado el test de aceptación.
 - Se puede llevar a cabo la conversación aunque no esté presente alguna tarea involucrada en ella.

El principal inconveniente de las conversaciones es que el error en uno de los procesos produce que todos deban retroceder. Aunque tiene la ventaja de que puede tomar caminos alternativos ya que se recupera el estado consistente.

- Recuperación de estados hacia adelante: si se lanza una excepción en alguno de los procesos de las acciones atómicas se lanza la excepción en todas.

Se presenta el problema de las excepciones concurrentes. El lanzarse dos o más excepciones concurrentes no se define bien cual es el manejador que se debe ejecutar.

Una posible solución es un árbol de excepciones, dónde se elige la excepción raíz del subárbol que contiene todas las excepciones que se han lanzado.

Otro mecanismo que permite la interacción entre las tareas y la recuperación de errores en las acciones atómicas es la **notificación asíncrona**. Permite que una tarea llame a la otra sin tener que esperar. Surgen así las dos vías de tratamiento de excepciones:

- Reanudación (manejo de eventos): cada proceso indica las excepciones que maneja. En caso de que se lanza una se avisa y se cambia el flujo de control del proceso al manejador.
- Terminación: cada proceso determina la transferencia asíncrona de control que lo puede terminar.

Control de recursos

Como continuación del tema anterior, en este tema se tratan los procesos competitivos. Un tema importante en este caso es la distinción entre sincronización condicional y sincronización evitable en el modelo de concurrencia.

Control de los recursos

Aunque los procesos son independientes entre sí, el acto de asignación de recursos puede provocar problemas si existe algún error en los procesos (inanición o interbloqueo).

La comunicación es necesaria para la asignación de recursos. En los lenguajes de programación los recursos deberían encapsularse y se debe acceder a ellos a través de interfaces de alto nivel.

El monitor representa una buena opción para regular las condiciones de comunicación para la asignación de recursos.

Aproximaciones lingüísticas para acceso a un servicio o recurso:

- Espera condicional: se aceptan todas las solicitudes, las que no se pueden ejecutar se ponen en una cola de espera (monitor).
- Evitación: solo se aceptan aquellas peticiones que se pueden satisfacer, comprobable mediante guardas. Un problema que presentan los monitores en esta opción es que la cola de espera se maneja de forma arbitraria.

Puede haber procesos que se queden siempre en la cola de espera y por no llegar a obtener el recurso nunca se ejecuten.

Inanición

Interbloqueo o bloqueo mutuo

También conocido como deadlock se trata del bloqueo permanente de un conjunto de procesos o hilos de ejecución que compiten o se comunican entre ellos en un sistema concurrente. No existe una solución general para los interbloqueos.

Es posible representar bloqueos mutuos mediante grafos dirigidos, el proceso es representado por un cuadrado y el recurso por un círculo. Cuando un proceso solicita un recurso se dibuja una flecha dirigida desde el proceso al recurso, cuando el recurso está asignado a un proceso la flecha está dirigida desde el recurso al proceso.

Existen cuatro condiciones necesarias para que se de un interbloqueo, estas son:

- Condición de exclusión mutua, existe un recurso compartido por los procesos al que solo puede acceder un proceso simultaneamente.
- Condición de retención y espera, al menos un proceso P ha adquirido un recurso R y lo retiene mientras espera un recurso R2 que ha sido asignado a otro proceso.
- Condición de no expropiación, los recursos no pueden ser expropiados por otros procesos, tienen que ser liberados por sus propietarios.
- Condición de espera circular, dado un conjunto de proceso P0..PM, P0 esta esperando un recurso adquirido por P1, que esta esperando un recurso adquirido por P2, ..., que esta esperando un recurso adquirido por PM, que esta esperando un recurso adquirido por P0. Esta condición implica la condición de retención y espera.

Hay tres perspectivas posibles para tratar con los interbloqueos:

- La prevención.
- Evitación.
- Detección y recuperación.

Capacidades de tiempo real

Los requisitos temporales constituyen la característica diferenciadora de los sistemas de tiempo real. En este tema se presenta una visión de estos requisitos y de las funcionalidades del lenguaje y estrategias de implementación que se utilizan para satisfacerlos. Los sistemas de tiempo real estrictos tienen restricciones de tiempo que deben ser satisfechas, los sistemas no estrictos fallan a veces a la hora de cumplir dichas restricciones adecuadamente. Los dos casos se consideran en el contexto de la planificación con tiempos límite.

Tras el tema anterior de capacidades de tiempo real, hay que incluir estas capacidades en la planificación de tareas, introduciéndose también la noción de prioridad junto con el análisis de la planificabilidad para sistemas con desalojo basado en prioridad.

Vamos a barajar tres modelos de planificación:

- Modelo simple
- Ejecutivo cíclico
- Planificación basada en procesos

Modelo simple

Tenemos un modelo que permite describir algunos esquemas de planificación estándar:

- La aplicación está compuesta por un conjunto fijo de procesos
- Los procesos son periódicos, con periodos conocidos
- Los procesos son independientes entre sí
- El instante crítico es cuando todos los procesos son ejecutados a la vez
- Las sobrecargas del sistema, tiempos de cambio de contexto y demás se suponen con coste cero
- Los tiempos límite de los procesos son iguales a sus periodos
- Los procesos tienen tiempo de ejecución constante en el peor caso

Al considerar que hay independencia entre procesos se puede suponer que en algún instante de tiempo todos los procesos son ejecutados a la vez, esto representará la carga máxima para el procesador y se conocerá como *instante crítico*.

Para definir el modelo de proceso simple definimos las siguientes variables:

Notación	Descripción
B	Tiempo de bloqueo en el peor caso
C	Tiempo de ejecución del proceso en el peor caso (WCET)
D	Tiempo límite del proceso
I	Tiempo de interferencia del proceso
J	Fluctuación en la ejecución del proceso
N	Número de procesos en el sistema
P	Prioridad asignada al proceso
T	Tiempo mínimo entre ejecuciones del proceso (periodo)
Y	Utilización de cada proceso (C/T)
a-z	Nombre del proceso

Ejecutivo cíclico

Una forma común de la implementación de sistemas de tiempo real es el uso de un ejecutivo cíclico.

El ejecutivo cíclico es una tabla de llamadas a procedimientos, donde cada procedimiento representa parte del código de un proceso. A la tabla completa se le conoce como *ciclo principal* y habitualmente consta de cierto número de *ciclos secundarios*, cada uno de ellos de duración fija. De esta forma, por ejemplo, cuatro ciclos secundarios de 25 ms de duración conforman un ciclo principal de 100 ms. Durante la ejecución, una interrupción de reloj cada 25 ms permite que el planificador realice rondas entre los cuatro ciclos secundarios.

Existe una tabla de llamadas a procedimiento con un ciclo principal y ciclos secundarios de duración fija.

Las propiedades del ejecutivo cíclico son:

- Los procesos no existen en tiempo de ejecución, solamente existe el proceso principal
- Los procedimientos comparten un espacio de direcciones, pueden compartir datos sin necesidad de protección
- Los periodos deben ser múltiplos del tiempo de ciclo secundario

El ejecutivo cíclico presenta los siguientes inconvenientes:

- Dificultad para incorporar procesos esporádicos
- Dificultad para incorporar procesos con periodos grandes, el tiempo del ciclo mayor es el único que se puede usar sin replanificación
- Dificultad para construir el ejecutivo cíclico
- Procesos con tiempo notable tendrán que ser divididos en procedimientos de tamaño fijo, es propenso a errores

Si puede construirse un ejecutivo cíclico no será necesario ningún test de planificabilidad más. El problema que plantea el ejecutivo cíclico es que a partir de cierto tamaño se convierte en irresoluble, para sistemas periódicos simples se mantiene como una estrategia de implementación apropiada.

Planificación basada en procesos

Una alternativa al enfoque del ejecutivo cíclico (la ejecución consiste en una secuencia de llamadas a procedimientos) es soportar de forma directa la ejecución de procesos y determinar cual es el proceso que deberá ejecutarse en cada instante de tiempo mediante uno o más atributos de planificación. Un proceso está limitado a estar en uno de los posibles estados siguientes:

- Ejecutable
- Suspendido en espera de un evento temporizado (apropiado para procesos periódicos)

- Suspendido en espera de un evento no temporizado (apropiado para procesos esporádicos)

Alternativas de planificación

Existe un gran número de aproximaciones distintas a la planificación:

- Planificación de prioridad estática o fija: FPS
- Primero el tiempo límite más temprano: EDF
- Planificación basada en el valor: VBS

Respecto a la apropiación de procesos, por derecho preferente existen estas opciones:

- Esquema apropiativo, Cambio de procesos inmediato
- Esquema no apropiativo, se esperará a la finalización del proceso de menor prioridad
- Esquema de apropiación diferida o de distribución cooperativa, tiempo limitado para el proceso de menor prioridad

Planificación de prioridad de tasa monotónica

A cada proceso se le asigna una prioridad en función de su periodo, los procesos con menor periodo tienen la mayor prioridad.

Test de planificabilidad basada en la utilización

El sumatorio de los tiempos de uso de CPU de cada proceso tiene que ser menor que el valor de la ecuación del test de planificabilidad, lo cual asegura que el conjunto de procesos es planificable, que sea mayor no implica que el conjunto de procesos no sea planificable.

El test supone una condición suficiente pero no necesaria.

$$U \equiv \sum_{i=1}^N \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1)$$

N=3 U<=0.78

Test de utilización para EDF

Primero el proceso con tiempo límite más cercano.

- Es más simple

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq 1$$

- Es más sencillo de implementar
- Más sencillo incorporar procesos sin tiempos límites en FPS
- Es más fácil trabajar con prioridades (FPS) que con deadlines (EDF)
- FPS es más predecible en situaciones de sobrecargas

Protocolos de acotación de la prioridad

Se trata de unos protocolos diseñados para minimizar las situaciones de cadenas de bloqueo y eliminar condiciones de fallo.

Existen dos tipos:

- Protocolo original de acotación de la prioridad: OCPP
- Protocolo inmediato de acotación de la prioridad: ICPP

Cuando se utilizan estos protocolos en un sistema monoprocesador se cumplen las siguientes propiedades:

- Un proceso de alta prioridad puede ser bloqueado por procesos de prioridad baja en una sola ocasión como máximo.
- Se previenen los bloqueos mutuos
- Se previenen los bloqueos transitivos
- Se aseguran los accesos mutuamente excluyentes a recursos

El protocolo asegura que si un recurso está bloqueado por cierto proceso a y esto conduce a que se bloquee un proceso de mayor prioridad, b, entonces no se permite que ningún otro recurso que pueda bloquear a b sea bloqueado más que por a. Un proceso puede ser retardado no sólo mientras esta intentando bloquear un recurso previamente bloqueado, sino también cuando ese bloqueo pudiera producir un bloqueo múltiple de procesos de mayor prioridad.

Protocolo original de acotación de la prioridad

El protocolo original toma la siguiente forma:

- Cada proceso tiene asignada una prioridad estática por defecto
- Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo están utilizando
- Un proceso tiene una prioridad dinámica que es el máximo de su prioridad estática y de cualquiera de las que herede debido a bloqueos
- Un proceso sólo puede bloquear un recurso si su prioridad dinámica es mayor que la cota máxima de cualquier recurso actualmente bloqueado (excluyendo los bloqueados por él)

Se permite el bloqueo del primer recurso del sistema. El efecto del protocolo es asegurar que el segundo recurso sólo pueda ser bloqueado si no existe un proceso de mayor prioridad que utilice ambos recursos. Consecuentemente, la cantidad máxima de tiempo que un proceso puede ser bloqueado es igual al tiempo de ejecución de la sección crítica más larga en cualquiera de los procesos de menor prioridad que son accedidos por procesos de alta prioridad.

Protocolo inmediato de acotación de la prioridad

El algoritmo inmediato de acotación de la prioridad (ICPP) toma un enfoque más sencillo, y fija la prioridad de un proceso tan pronto bloquea un recurso (en lugar de hacerlo cuando realmente bloquea a un proceso de mayor prioridad). El protocolo se define como sigue:

- Cada proceso tiene asignada una prioridad por defecto (quizás mediante el esquema monotónico de tiempo límite).
- Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo utilizan.
- Un proceso tiene una prioridad dinámica, que es el máximo entre su propia prioridad estática y los valores techo de cualquier recurso que tenga bloqueado.

Como consecuencia de esta última regla, un proceso sólo podrá ser bloqueado al principio de su ejecución. Una vez que el proceso comience a ejecutarse, todos los recursos necesarios debería estar libres, si no lo estuvieran, entonces algún proceso tendría una prioridad mayor o igual, y la ejecución del proceso deberá ser pospuesta.

Aunque el comportamiento en el peor caso de los dos esquemas de acotación es idéntico existen algunas diferencias:

- ICCP es más sencillo de implementar que el original (OCP), ya que no hay que monitorizar las relaciones de bloqueo.
- ICCP produce menos cambios de contexto, ya que el bloqueo es previo a la primera ejecución.
- ICCP requiere más cambios de prioridad, ya que éstos se producen con todas las utilizaciones de recursos, OCP modifica la prioridad sólo si se producen bloqueos.

CAPÍTULO 11

Programación de bajo nivel

Un requisito importante de muchos sistemas de tiempo real es que incorporan dispositivos externos que deben ser programados (controlados) como una parte del software de la aplicación. Esta programación a bajo nivel no concuerda con la aproximación abstracta para la producción de software que caracteriza a la ingeniería del software. En este tema se considera las formas en que las funcionalidades de bajo nivel pueden ser incorporadas con éxito en los lenguajes de alto nivel.

Problemas de planificabilidad

Algoritmo de planificación de tasa monotónica

Tres procesos lógicos (P, Q y S) tienen las siguientes características:

- P: periodo 3, tiempo de ejecución necesario 1.
- Q: periodo 6, tiempo de ejecución necesario 2.
- S: periodo 18, tiempo de ejecución necesario 5.

Explique el algoritmo de planificación de tasa monotónica (rate monotonic scheduling algorithm) y muestre cómo pueden planificarse estos procesos utilizando el algoritmo de planificación de tasa monotónica.

El algoritmo de planificación de tasa monotónica se basa en asignar la prioridad en función del periodo del proceso, de forma que los procesos con menor periodo tendrán la mayor prioridad.

De esta forma en el caso del enunciado del problema la tabla de prioridades quedara de la forma:

Proceso	Periodo (T)	Tiempo de ejecución (C)	Prioridad(P)
P	3	1	3
Q	6	2	2
S	18	5	1

Una vez asignada la prioridad a los procesos escribimos en una tabla las fracciones de tiempo asignadas a cada proceso

I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P	X			X			X			X			X			X			X
Q		X	X					X	X					X	X				
S					X	X					X	X					X		

Como observamos en la tabla en un fragmento de 18 segmentos de tiempo los tres procesos se han podido planificar por lo que podemos deducir que este conjunto de procesos es planificable.

Test simple de planificabilidad para FPS

¿Es planificable el conjunto de procesos mostrado en la tabla? (utilice el test simple de planificabilidad para FPS)

Proceso	Periodo (T)	Tiempo de Ejecución (C)
a	50	10
b	40	10
c	30	9

El primer paso es establecer la prioridad de los procesos en base al periodo (menor periodo, mayor prioridad):

Proceso	Periodo (T)	Tiempo de ejecución (C)	Prioridad (P)
a	50	10	3
b	40	10	2
c	30	9	1

Una vez asignada la prioridad a los procesos calculamos el tiempo de uso de CPU de cada proceso:

Proceso	T	C	P	Tiempo de uso de CPU (C/T)
a	50	10	3	20 %
b	40	10	2	25 %
c	30	9	1	30 %

Para realizar el test simple de planificabilidad utilizamos la siguiente formula, siendo N el número de procesos a planificar:

$$U \equiv \sum_{i=1}^N \frac{C_i}{T_i} < N * (2^{1/N} - 1)$$

De esta forma para el numero de procesos del ejercicio (3) tenemos el valor:

$$3 * (2^{1/3} - 1) = 0,78$$

Sumando el tiempo de uso de los tres procesos tenemos el valor 0,75, como $0,78 > 0,75$ podemos afirmar que el conjunto de procesos es planificable.

Estudio de planificación

Estudie la planificación del siguiente conjunto de procesos:

Proceso	Periodo (T)	Tiempo de Ejecución (C)
A	75	35
B	40	10
C	20	5

El primer paso es establecer la prioridad de los procesos en base al periodo (menor periodo, mayor prioridad) y el tiempo de utilización:

Proceso	T	C	P	Tiempo de uso de CPU (C/T)
A	75	35	1	46,6 %
B	40	10	2	25 %
C	20	5	3	25 %

Para realizar el test simple de planificabilidad utilizamos la siguiente formula, siendo N el número de procesos a planificar:

$$N * (2^{1/N} - 1)$$

De esta forma para el numero de procesos del ejercicio (3) tenemos el valor:

$$3 * (2^{1/3} - 1) = 0,78$$

El caso mostrado no supera el test basado en la utilización ya que $0,78 < 0,96$, el umbral de utilización es muy superior al tiempo limite de $0,78$ por lo cual no podemos afirmar que el conjunto de procesos sea planificable en función del test de planificabilidad, aunque esto no es condición suficiente para afirmar que el conjunto de procesos no es planificable.

Si optamos por analizar los tiempos de respuesta para FPS, el proceso de mayor prioridad tiene un tiempo de respuesta:

$$\begin{aligned} R_i &= C_i \\ R_C &= 5 \end{aligned}$$

El resto de procesos sufren interferencias:

$$R_i = C_i + I_i$$

El valor máximo de la interferencia viene dado por: $\sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j$

T_j es el periodo de aquellos procesos j con mayor prioridad que i .

$$\begin{aligned} R_c &= C_c = 5 \\ w_b^0 &= C_b = 10 \\ w_b^1 &= C_b + \lceil \frac{w_b^0}{T_c} \rceil C_c = 10 + \lceil \frac{10}{20} \rceil 5 = 15 \\ w_b^2 &= C_b + \lceil \frac{w_b^1}{T_c} \rceil C_c = 10 + \lceil \frac{15}{20} \rceil 5 = 15 \\ R_b &= 15 \\ w_a^0 &= C_a = 35 \\ w_a^1 &= C_a + \lceil \frac{w_a^0}{T_c} \rceil C_c + \lceil \frac{w_a^0}{T_b} \rceil C_b = 35 + \lceil \frac{35}{20} \rceil 5 + \lceil \frac{35}{40} \rceil 10 = 55 \\ w_a^2 &= C_a + \lceil \frac{w_a^1}{T_c} \rceil C_c + \lceil \frac{w_a^1}{T_b} \rceil C_b = 35 + \lceil \frac{55}{20} \rceil 5 + \lceil \frac{55}{40} \rceil 10 = 70 \\ w_a^3 &= C_a + \lceil \frac{w_a^2}{T_c} \rceil C_c + \lceil \frac{w_a^2}{T_b} \rceil C_b = 35 + \lceil \frac{70}{20} \rceil 5 + \lceil \frac{70}{40} \rceil 10 = 75 \\ w_a^4 &= C_a + \lceil \frac{w_a^3}{T_c} \rceil C_c + \lceil \frac{w_a^3}{T_b} \rceil C_b = 35 + \lceil \frac{75}{20} \rceil 5 + \lceil \frac{75}{40} \rceil 10 = 75 \\ R_a &= 75 \end{aligned}$$

Para sacar los valores se redondea el contenido de la fracción al entero más cercano, en el momento en que encontramos los valores de $w_a^3 = w_a^4$ es cuando podemos comprobar si el tiempo de respuesta cumple los requerimientos.

En el caso de estudio el problema es planificable. Una ventaja de este método es que el calculo de los tiempos de respuesta es condición suficiente y necesaria para decidir si el conjunto de procesos es planificable.

Preguntas de exámenes

Mecanismos cambio de contexto

Entre los mecanismos necesarios para permitir la entrada y salida dirigidas por interrupciones están los mecanismos de cambio de contexto. En el espacio de una cara comente estos mecanismos de cambio de contexto.

Cuando ocurre una interrupción, debe preservarse el estado actual del procesador y activarse la rutina de servicio correspondiente. Una vez servida la interrupción, habrá que retomar el proceso y continuar su ejecución. Otra posibilidad es que el planificador seleccione un nuevo proceso como consecuencia de la interrupción. El procedimiento completo se conoce como cambio de contexto y sus acciones pueden resumirse como sigue:

- Preservación del estado del procesador inmediatamente anterior a la aparición de la interrupción.
- Establecimiento del procesador en el estado adecuado para procesar la interrupción.
- Restauración del estado del proceso suspendido, tras completar el procesamiento de la interrupción.

El estado de un proceso en ejecución sobre un procesador consta de:

- La posición en memoria de la instrucción actual (o la siguiente) en la secuencia de ejecución.
- La información de estatus del programa (que podrá contener información relativa al modo de procesamiento, a la prioridad actual, a la protección de memoria, a las interrupciones permitidas y otras).
- Los contenidos de los registros programables.

El tipo de cambio de contexto provisto se caracteriza por la capacidad del hardware de preservar y restaurar el estado del proceso. Se pueden distinguir tres niveles de cambio de contexto:

- Básico: sólo se guarda y se actualiza el contador de programa
- Parcial: se guardan el contador de programa y los registros de estatus del programa para alojar los nuevos valores.
- Completo: se guarda el contexto completo del proceso y se aloja uno nuevo.

Dependiendo del grado necesario de preservación del estado, puede que haya que complementar las acciones del hardware con soporte explícito de software. Por ejemplo, puede que un cambio parcial de contexto sea adecuado para un modelo de manejo de interrupciones que contemple al manejador como un procedimiento o subrutina. Sin embargo, si el manejador es visto como un proceso separado con sus propias áreas de pila y de datos, será necesario un manejador

de bajo nivel que se haga cargo completamente del cambio de contexto. Si, por otra parte, el hardware puede con un cambio de contexto completo, dicho manejador de bajo nivel no será necesario. La mayoría de los procesadores sólo proporcionan un cambio parcial de contexto. Sin embargo, el procesador ARM proporciona una interrupción rápida, donde se guardan también algunos de los registros de propósito general.

Hay que hacer constar que algunos procesadores modernos permiten la búsqueda, decodificación y ejecución de instrucciones en paralelo. Algunos incluso permiten la ejecución de instrucciones al margen de la secuencia especificada en el programa. Este capítulo parte de la premisa de que las interrupciones son precisas en el sentido de que, cuando se ejecuta un manejador de interrupción:

- Todas las instrucciones que ha tomado el procesador antes de la instrucción interrumpida han finalizado su ejecución y han modificado correctamente el estado del programa.
- Aquellas instrucciones tornadas por el procesador tras la instrucción interrumpida han sido ejecutadas y no han modificado el estado del programa.
- Si fue la misma instrucción la que causó la interrupción (por ejemplo, provocando una violación de memoria), dicha instrucción ha sido ejecutada totalmente o no se ha ejecutado en absoluto.

Si una interrupción no es precisa, se supone que la recuperación es transparente al software manejo de la interrupción.

Fallos de temporización y valor

Explique como se puede transformar un sistema de forma que todos los fallos de temporización se manifiesten como fallos de valor. ¿Se puede conseguir tal conversión?

Un fallo de temporización se define como la entrega de un servicio fuera de su intervalo de administración definido, por lo general más allá de una fecha límite definida. A menudo el servicio se entrega tarde debido al tiempo necesario para construir el valor correcto para el servicio. Si el sistema se diseñó para siempre entregar un valor en el intervalo correcto, entonces la falta de tiempo se manifestaría como un valor incorrecto. De ahí que los fallos de sincronización y el valor no pueden ser considerados ortogonales.

Lo contrario de lo anterior también puede ser cierto. Un servicio que ha fracasado porque el valor que ofrece es incorrecto, puede ser capaz de entregar un valor correcto si se administra más CPU de tiempo; por lo tanto, un valor correcto puede ser entregado pero demasiado tarde. Sin embargo, esto no es una verdad universal, un fallo de valor (o error) puede ser debido a muchas razones distintas de insuficiente asignación de ciclos de procesador.

Primitivas de sincronización

En la cara de una hoja comente los conceptos de potencia expresiva y facilidad de uso de las primitivas de sincronización para control de recursos.

Bloom ha sugerido criterios para evaluar primitivas de sincronización en el contexto de la gestión de recursos. Este análisis forma la base de esta sección, donde se consideran la potencia expresiva y la facilidad de uso de las primitivas de sincronización para control de recursos. Las primitivas a evaluar son monitores/métodos sincronizados (con su uso de sincronización de condición), servidores (con una interfaz de paso de mensajes) y recursos protegidos (implementados como objetos protegidos). Los dos últimos utilizan guardas para la sincronización y por ello un aspecto de este análisis es una comparación entre sincronización de condición y sincronización de evitación.

La expresión **potencia expresiva** se utiliza para indicar la capacidad de un lenguaje para expresar las restricciones de sincronización requeridas. La **facilidad de uso** de una primitiva de sincronización abarca:

- La facilidad con la que expresa cada una de esas restricciones de sincronización.
- La facilidad con la que se permite combinar las restricciones para conseguir esquemas de sincronización más complejos.

En el contexto del control de recursos, la información necesaria para expresar estas restricciones se puede clasificar como sigue:

- Tipo de petición de servicio.
- Orden en el que llegan las solicitudes.
- Estado del servidor y de todos los objetos que gestiona.
- Parámetros de una solicitud.

El conjunto original de restricciones incluía la **historia del objeto** (esto es, la secuencia de todas las solicitudes previas). Aquí se supone que el estado del objeto se puede extender para incluir la información histórica. Además, se añade a la lista:

- La prioridad del cliente

Se considera la prioridad como medida de la importancia del proceso.

Hay en general dos aproximaciones lingüísticas a la restricción del acceso para un servicio. La primera es la espera condicional, se aceptan todas las solicitudes pero cualquier proceso cuya solicitud no se puede atender se suspende en una cola. El monitor convencional tipifica esta aproximación: un proceso cuya solicitud no se puede atender es encolado en una variable de condición y reanudado cuando se puede atender la solicitud. La segunda aproximación es evitación: las solicitudes no se aceptan a menos que puedan ser satisfechas. Las condiciones bajo las que se puede aceptar una solicitud de forma segura se expresan como una guarda en una acción de aceptación.

Técnicas de detección de errores

La efectividad de cualquier sistema tolerante a fallos depende de la efectividad de sus técnicas de detección de errores. Describa las dos clases de técnicas de detección de errores.

- **DETECCIÓN EN EL ENTORNO:** Los errores se detectan en el entorno en el cual se ejecuta el programa. Se incluye aquellos detectados por el hardware, como los de *ejecución de instrucción ilegal*, *desbordamiento aritmético*, o *violación de protección*. También son considerados los errores detectados en tiempo de ejecución por el sistema soporte del lenguaje de programación de tiempo real; por ejemplo, los de *error en los límites del array*, *referencia a apuntador nulo*, o *valor fuera de rango*.
- **DETECCIÓN EN LA APLICACIÓN:** Los errores se detectan por la aplicación misma. La mayoría de las técnicas que se pueden utilizar en la aplicación corresponden a alguna de las siguientes categorías:
 - Comprobación de réplicas. Se ha demostrado que la programación de N-Versiones puede ser utilizada para tolerar fallos software, y también como técnica para la detección de errores (utilizando una redundancia de 2-Versiones)
 - Comprobaciones temporales. Existen dos tipos de comprobaciones temporales. El primer tipo implica un proceso temporizador guardián, que sino es puesto a cero por un cierto componente dentro de un cierto periodo de tiempo, se supone que dicho componente esta en un estado de error. En los sistemas embebidos, donde los tiempos de respuesta son importantes, se necesita un segundo tipo de comprobación. De esta manera se detectan fallos asociados con el incumplimiento de tiempos límite.
 - Comprobaciones inversas. Estas son posibles en componentes donde exista una relación uno a uno entre la entrada y la salida.
 - Códigos de comprobación. Los códigos de comprobación se utilizan para comprobar la corrupción de los datos.
 - Comprobaciones de racionalidad. Se basan en el conocimiento del diseño y de la construcción del sistema. Comprueban que el estado de los datos o el valor de un objeto es razonable basandose en su supuesto uso.

- Comprobaciones estructurales. Las comprobaciones estructurales son utilizadas para comprobar la integridad de los objetos de datos tales como listas o colas. Podrían consistir en contar el número de elementos en el objeto, en apuntadores redundantes o en información extra sobre su estatus.
- Comprobaciones de racionalidad dinámica. En la salida producida por algunos controladores digitales, habitualmente existe una relación entre cualesquiera dos salidas consecutivas. Por lo tanto, se podrá detectar un error si el valor de una salida nueva difiere considerablemente del valor de la salida anterior.

Acciones atómicas

Distinguir entre una acción atómica y una transacción atómica, ¿Cuál es la relación entre una transacción atómica y una conversación?

La expresión transacciones atómicas se ha utilizado frecuentemente en el marco conceptual de los sistemas operativos y las bases de datos. Una transacción atómica tiene todas las propiedades de una acción atómica, más la característica adicional de que su ejecución puede tener éxito o fallar (no éxito). Por fallo se entiende la ocurrencia de un error del que la transacción no puede recuperarse, por ejemplo, un fallo de procesador. Si falla una acción atómica, los componentes del sistema que están siendo manipulados por la acción pueden terminar en un estado inconsistente.

Ante un fallo una transacción atómica garantiza que los componentes son devueltos a su estado original, esto es, al estado en el que estaban antes de comenzar la transacción, las transacciones atómicas a veces se conocen con el nombre de acciones recuperables, aunque desafortunadamente, se tiende a confundir los términos acción atómica y transacción atómica.

Las dos propiedades distintivas de las transacciones atómicas son:

- Atomicidad de fallos, lo que significa que la transacción debe o bien ser completada con éxito, o (en el caso de fallar) no tener efecto.
- Atomicidad de sincronización, o aislamiento, lo que significa que la transacción es indivisible, en el sentido de que su ejecución parcial no puede ser observada por ninguna transacción que se esté ejecutando concurrentemente.

A pesar de que las transacciones atómicas son útiles para las aplicaciones que implican la manipulación de bases de datos, no son adecuadas para la programación de sistemas tolerantes a fallos per se. Esto se debe a que precisan de algún tipo de mecanismo de recuperación proporcionado por el sistema. Este mecanismo viene preestablecido, sin que el programador tenga control sobre su operativa. A pesar de que las transacciones atómicas proporcionan una forma de recuperación de errores hacia atrás, no permiten la escritura de procedimientos de recuperación. Independientemente de lo anterior, las transacciones atómicas tienen su sitio en la protección de la integridad en los sistemas de bases de datos de tiempo real.

(No esta bien contestada)

Manejo de excepciones

En los mecanismos de manejo de excepciones, si el manejador resolviera el problema que causó la generación de la excepción, sería posible que reanudara su trabajo. Esto se conoce como modelo de reanudación, describa como funciona dicho modelo

Para ilustrar el modelo de reanudación, considere tres procedimientos (P, Q y R). El procedimiento P invoca Q, que a su vez invoca R. El procedimiento R genera una excepción (r) que es manejada por Q, asumiendo que no hay un manejador local en R. El manejador para (r) es Hr, mientras maneja (r), Hr genera la excepción q que es manejada por Hq en el procedimiento P. Una vez manejada q, Hr continua su ejecución, y despues continua R.

El modelo de reanudación se entiende mucho mejor si contemplamos el manejador como un procedimiento que se invoca implícitamente al generar la excepción.

El problema de esta aproximación es la dificultad a la hora de reparar errores generados por el entorno de ejecución. Por ejemplo, un desbordamiento aritmético en medio de una secuencia compleja de expresiones podría ocasionar que varios de los registros contuvieran evaluaciones parciales. Al llamar al manejador, es probable que se sobrescriban dichos registros.

Tanto el lenguaje Perl como el lenguaje Mesa proporcionan un mecanismo que permite al manejador volver al contexto donde se generó la excepción. Ambos lenguajes soportan también el modelo de terminación.

Cuando realmente se nota la ventaja del modelo de reanudación es cuando la excepción ha sido generada asincrónicamente y por lo tanto tiene poco que ver con la ejecución actual del proceso.

Interbloqueos

Estudie si este sistema se encuentra en una situación de interbloqueo, y explique sus razones.

Considérese un sistema que tiene cinco recursos (P1, P2, P3, P4, P5) y siete recursos (R1, R2, R3, R4, R5, R6, R7). Hay una instancia de los recursos 2, 5 y 7, y dos instancias de los recursos 1, 3, 4 y 6. El proceso 1 tiene asignada una instancia de R1 y requiere una de R7. El proceso 2 tiene asignadas una instancia de R1, R2 y R3 y requiere una de R5. El proceso 3 tiene asignada una instancia de R3 y R4 y requiere una de R1. El proceso 4 tiene asignada R4 y R5 y requiere una de R2. El proceso 5 tiene una de R7.

Para que el proceso se encuentre en una situación de interbloqueo debe cumplir cuatro condiciones:

- *Condición de exclusión mutua*, existe un recurso compartido por los procesos al que solo puede acceder un proceso simultáneamente, los recursos 2, 5 y 7 cumplen esta condición.
- *Condición de retención y espera*, al menos un proceso P ha adquirido un recurso R y lo retiene mientras espera un recurso R2 que ha sido asignado a otro proceso. El proceso 4 tiene adquirido el recurso R5 mientras espera el recurso R2 que está bloqueado por el proceso 2
- *Condición de no expropiación*, los recursos no pueden ser expropiados por otros procesos, tienen que ser liberados por sus propietarios.
- *Condición de espera circular*, el proceso P2 tiene bloqueada la única instancia del recurso R2, requiere a su vez la instancia del recurso R5 que está bloqueado por el proceso 4 que a su vez requiere el recurso R2 por lo que se produce una situación de retención y espera.

Protocolos de acotación de prioridad

Sobre los protocolos de acotación de prioridad (priority ceiling protocols), responder a las siguientes cuestiones:

1. ¿Qué cuestiones abordan los protocolos de acotación de la prioridad?
 2. ¿Qué forma toma el protocolo original de acotación de la prioridad?
 3. ¿Cómo se define el protocolo inmediato de acotación de la prioridad?
 4. Aunque el comportamiento en el peor de los casos de los dos esquemas de acotación es idéntico (desde el punto de vista de la planificación), existen diferencias, indicar cuáles son.
1. Aunque el protocolo estándar de herencia da un límite superior para el número de bloqueos con los que se puede encontrar un proceso de prioridad alta, este límite puede todavía conducir a un cálculo del peor caso inaceptablemente pesimista. Esto se debe a la posibilidad de desarrollar cadenas de bloqueos (bloqueos transitivos), es decir, situaciones en las que el proceso c es bloqueado por el proceso b, el cual está bloqueado por el proceso a, y así sucesivamente. Como los datos compartidos son un recurso del sistema, desde el punto de vista del gestor de recursos no sólo se debe minimizar el bloqueo, sino que las condiciones de fallo (como los interbloqueos) deben

ser eliminadas. Los protocolos de acotación de la prioridad abordan todas estas cuestiones. Consideramos dos de ellos:

- Protocolo original de acotación de la prioridad.
- Protocolo inmediato de acotación de la prioridad

Cuando se utiliza cualquiera de estos protocolos en un sistema monoprocesador:

- Un proceso de alta prioridad puede ser bloqueado por procesos de prioridad baja en una sola ocasión como máximo durante su ejecución.
- Se previenen los bloqueos mutuos (interbloqueos).
- Se previenen los bloqueos transitivos.
- Se aseguran (por el protocolo mismo) los accesos mutuamente excluyentes a los recursos.

La mejor manera de describir los protocolos de acotación de la prioridad es en relación con los recursos protegidos por secciones críticas. En esencia, el protocolo asegura que si un recurso está bloqueado por cierto proceso *a*, y esto conduce a que se bloquee un proceso de mayor prioridad *b*, entonces no se permite que ningún otro recurso que pueda bloquear a *b* sea bloqueado más que por un *a*. Un proceso, por lo tanto, puede ser retardado no sólo mientras está intentando bloquear un recurso previamente bloqueado, sino también cuando ese bloqueo pudiera producir un bloqueo múltiple de procesos de mayor prioridad.

2. El protocolo original toma la siguiente forma:

- a) Cada proceso tiene asignada una prioridad estática por defecto (quizás según el esquema monotónico de tiempo límite).
- b) Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo están utilizando.
- c) Un proceso tiene una prioridad dinámica que es el máximo de su prioridad estática y de cualquiera de las que herede debido a que bloquea procesos de mayor prioridad.

Se permite el bloqueo del primer recurso del sistema. El efecto del protocolo es asegurar que el segundo recurso solo pueda ser bloqueado si no existe un proceso de mayor prioridad que utilice ambos recursos.

3. El algoritmo inmediato toma un enfoque más sencillo, fija la prioridad de un proceso tan pronto bloquea un recurso, el protocolo se define como sigue:

- a) Cada proceso tiene asignada una prioridad por defecto.
- b) Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo utilizan.
- c) Un proceso tiene una prioridad dinámica, que es el máximo entre su propia prioridad estática y los valores techo de cualquier recurso que tenga bloqueado.

Como consecuencia de esta última regla, un proceso sólo podrá ser bloqueado al principio de su ejecución. Una vez comience a ejecutarse, todos los recursos necesarios debería estar libres, si no lo estuvieran algún proceso tendría una prioridad mayor o igual y la ejecución del proceso deberá ser pospuesta.

4. Existen estas diferencias:

- a) ICPP es más sencillo de implementar que el original (OCP), no hay que monitorizar las relaciones de bloqueo.
- b) ICPP produce menos cambios de contexto, el bloqueo es previo a la primera ejecución.
- c) ICPP requiere más cambios de prioridad, estos se producen con todas las utilizaciones de recursos, OCP modifica la prioridad sólo si se producen bloqueos.

Semántica del reencolado

El concepto que está detrás del reencolado es el de mover la tarea (que estaba tras una guardia o barrera) detrás de otra guarda. Considérese una persona esperando para entrar en una habitación. Una vez dentro, la persona puede ser arrojada (reencolada) de la habitación y ser colocada de nuevo detrás de una puerta (potencialmente cerrada).

Ada permite reencolados entre entradas de tarea y entradas de objetos protegidos. Un reencolado puede ser para un mismo entry, para otro entry de la misma unidad, o para otra unidad. Se permiten los reencolados de entradas de tarea (y viceversa). Sin embargo, el principal uso del reencolado es enviar la tarea invocadora a un entry diferente de la misma unidad en la que se ejecuto el reencolado.

Semántica: Es importante apreciar que reencolado no es una simple llamada. Si el procedimiento P llama al procedimiento Q, entonces, una vez que Q ha finalizado, el control se pasa de nuevo a P. Pero si el entry X reencola en la entry Y, el control no se pasa de nuevo a X. Una vez que se ha completado Y, el control pasa de nuevo al objeto que llamo a X. Por tanto, cuando un cuerpo entry o accept ejecuta un reencolado, ese cuerpo se completa.

Una consecuencia de esto es que, cuando se realiza un reencolado desde un objeto protegido a otro, entonces la exclusión mutua sobre el objeto original se abandona una vez que se ha encolado la tarea. Otras tareas que esperan para entrar en el primer objeto serán capaces de hacerlo. Sin embargo, un reencolado sobre el mismo objeto protegido mantendrá el bloqueo de exclusión mutua (si el entry objetivo esta abierto).

Ejecución concurrente de procesos

Describe los tres mecanismos básicos de representación de la ejecución concurrente de procesos (Task representation concurrent exception).

Hay tres mecanismos básicos para representar la ejecución concurrente: fork y join, cobegin y la declaración explícita de procesos. A veces se incluyen también las corrutinas como mecanismo para expresar la ejecución concurrente.

La instrucción fork (bifurca) indica que cierta rutina deberá comenzar a ejecutarse concurrentemente con quien ha invocado el fork. La instrucción join (reúne) permite al que invoca detenerse y por ende sincronizarse hasta la terminación de la rutina invocada.

Cobegin (o parbegin, o par) es una forma estructurada de denotar la ejecución concurrente de un conjunto de instrucciones. La instrucción cobegin termina cuando han terminado todas las instrucciones concurrentes. Cada instrucción S puede ser cualquiera de las construcciones permitidas en el lenguaje, incluyendo las asignaciones sencillas o las llamadas a procedimientos. De invocar algún procedimiento, podrán pasarse datos a los procesos invocados mediante los parámetros de la llamada. La instrucción cobegin podría incluir, a su vez, una secuencia de instrucciones donde apareciera cobegin y así conseguir una jerarquía de procesos.

Las corrutinas son como subrutinas, salvo que permite el paso explícito de control entre ellas de una forma simétrica en vez de estrictamente jerárquica. El control se transfiere de una corrutina a otra mediante una sentencia reanuda que incluye el nombre de la corrutina con la que se continúa. Cuando una corrutina realiza una reanudación deja de ejecutarse, pero guarda información del estado local, de forma que si, posteriormente otra corrutina la hace reanudar podrá retomar su ejecución.

Seguridad, fiabilidad y confiabilidad

En el contexto de la fiabilidad y tolerancia a fallos, describa y compare los terminos: Seguridad, fiabilidad y confiabilidad (Safety, Reliability and Dependibility)

La seguridad software se considera a menudo en términos de percances. Un percance es un evento no planeado o secuencias de eventos que pueden producir muerte, lesión, enfermedad laboral, daño de equipos o propiedades o nocividad en el medio ambiente. Aunque la fiabilidad y la seguridad suelen considerarse como sinónimos, existe una

diferencia en el énfasis. La fiabilidad ha sido definida como la medida del éxito con el cual un sistema se ajusta a la especificación de su comportamiento. La fiabilidad ha sido definida como la medida del éxito con el cual un sistema se ajusta a la especificación de su comportamiento. Esto se expresa habitualmente en términos de probabilidad. La seguridad, sin embargo, es la improbabilidad de que se den las condiciones que conducen al percance, independientemente de si realiza la función prevista. Estas dos definiciones pueden entrar en conflicto.

De la misma manera que sucede con la fiabilidad, para cumplir con los requisitos de seguridad de un sistema embebido se debe realizar un análisis de seguridad a largo de todas las etapas de desarrollo de su ciclo de vida.

La noción de confiabilidad de un sistema es la propiedad del sistema que permite calificar, justificadamente, como fiable al servicio que proporciona. La confiabilidad por lo tanto incluye como casos especiales las nociones de fiabilidad y seguridad.

Modelos abstractos de manejo de dispositivos

Describe los modelos abstractos de manejo de dispositivos

Un dispositivo puede verse como un procesador que efectúa cierta tarea establecida, por lo que el sistema informático aparece como compuesto de varios procesos paralelos. Existen diversos modelos con los cuales el “proceso” del dispositivo podrá comunicarse y sincronizarse con los procesos en ejecución en el procesador principal. Todos ellos deben proporcionar:

1. Mecanismos para la representación, direccionamiento y manipulación de los registros de los dispositivos. Cada registro del dispositivo puede representarse como una variable del programa, un objeto, o incluso un canal de comunicación.
2. Una representación adecuada de las interrupciones.

En ella, se pueden encontrar diversas representaciones:

- Procedimiento: Se contempla la interrupción como una llamada a un procedimiento (en cierto sentido, es un procedimiento remoto invocado por el proceso del dispositivo). Cada comunicación y sincronización requerida deberá ser programada en el procedimiento de manejo de interrupción. El procedimiento no es anidado: solo podrá accederse al estado global o al estado local de manejador.
- Proceso esporádico: Se ve la interrupción como una petición de ejecución de cierto proceso. El manejador es un proceso esporádico, y puede acceder tanto a los datos persistentes locales como a los globales (si se dispone de un mecanismo de comunicación por variables compartidas en el modelo de concurrencia).
- Notificación asíncrona: Se ve la interrupción como una notificación asíncrona dirigida hacia un proceso. El manejador podrá acceder tanto al estado local del proceso como al estado global. Son posibles tanto el modelo de reanudación como el de terminación.
- Sincronización por variable de condición compartida: Se contempla la interrupción como una sincronización de condición dentro de un mecanismo de sincronización por variable compartida; por ejemplo, una operación de señal sobre un semáforo o una operación envía sobre una variable de condición en un monito. El manejador podrá acceder tanto al estado local del proceso/monitor como al estado global.
- Sincronización basada en mensajes: se contempla la interrupción como un mensaje vacío enviado por un canal de comunicación.

El proceso receptor podrá acceder al estado local del proceso.

Todas las posibilidades anteriores, excepto la aproximación procedimental, precisan de un cambio de contexto completo al ejecutar el manejador en presencia de un proceso. Si los manejadores cumplen ciertas restricciones, puede optimizarse el procedimiento. Por ejemplo, si el manejador en un modelo de notificación asíncrona presenta una semántica de reanudación y no accede a ningún dato local al proceso, bastaría un cambio parcial de contexto para manejar la interrupción.

Modelo sincronización de procesos

Describe la clasificación del modelo de sincronización de procesos por la semántica de la operación envía

En cualquier sistema basado en mensajes, un receptor no puede recibir un mensaje antes de que el emisor lo envíe. Con las variables compartidas, un lector no sabe si el escritor ha dejado un valor en ella.

Podemos tener variaciones en el modelo de sincronización de procesos, clasificándose así:

- Asíncrona: El emisor continúa independientemente del éxito o fracaso del envío.
- Síncrona: El emisor continúa una vez recibido el mensaje por el receptor.
- Invocación remota: El emisor continúa cuando el receptor devuelve una respuesta.

Se pueden usar dos eventos asíncronos para crear una relación síncrona y usar dos comunicaciones síncronas para construir una invocación remota.

El primero de estos dos casos tiene sus desventajas, como que se necesitan infinitos buffers para almacenar los mensajes no leídos (el receptor puede haber terminado). Además, la mayoría de envíos se programan para esperar un reconocimiento, se necesitan más comunicaciones con el método asíncrono (más complejidad) y es más difícil probar la corrección del sistema completo.

Mecanismos de control de E/S

Hay dos tipos de mecanismos para efectuar y controlar la E/S:

- Mecanismos de control dirigido por estatus.
- Mecanismos de control dirigido por interrupción.

Dirigido por estatus:

Cada programa realiza comprobaciones explícitas para terminar el estatus de un dispositivo dado. Una vez determinado el estatus del dispositivo, el programa podrá realizar las acciones pertinentes. Hay tres clases de instrucciones hardware para este tipo de mecanismo:

- Operaciones de test, que permiten al programa determinar el estatus de un dispositivo dado
- Operaciones de control, que indican al dispositivo que realice operaciones no relacionadas con transferencias (p.ej. posicionar las cabezas de lectura de un disco)
- Operaciones de E/S, que realizan la transferencia real de los datos entre el dispositivo y la UCP

Dirigidos por interrupción:

Incluso con el mecanismo dirigido por interrupción hay muchas variaciones posibles, dependiendo de cómo deban iniciarse y controlarse las transferencias. Tres variantes de este mecanismo son:

- Controlado por programa
- Iniciado por programa
- Controlado por programa de canal

Interbloqueos

Describe las condiciones necesarias para que se produzca un interbloqueo

Hay cuatro condiciones necesarias que se deben dar para que ocurra interbloqueo:

- **Exclusión mutua:** sólo un proceso puede utilizar un recurso al mismo tiempo (es decir, el recurso no se puede compartir o está limitado su acceso concurrente).
- **Mantenimiento y espera:** debe haber procesos que mantengan recursos mientras esperan por otros.
- **No desalojo (no apropiación):** un recurso sólo puede ser liberado por un proceso voluntariamente.
- **Espera circular:** debe existir una cadena circular de procesos, de forma que cada proceso mantenga recursos que son solicitados por el siguiente proceso en la cadena.

CAPÍTULO 14

Índices y tablas

- genindex

C

condición de carrera, 15

E

exclusión mutua, 13

P

Programación N-versiones, 5

S

sección crítica, 13

T

tasa monotónica, 35

test simple de planificabilidad, 36

tiempos de respuesta para FPS, 37