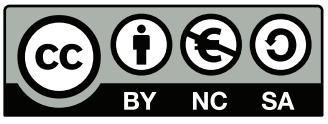
Sistemas Concurrentes y Distribuidos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM



Sistemas Concurrentes y Distribuidos

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1 SISTEMAS CONCURRENTES Y DISTRIBUIDOS

1.1 Tema 1. Introducción

1.1.1 1. Conceptos básicos y motivación

1.1 Modelos básicos relacionados con la concurrencia

- **Programa secuencial**: Declaraciones de datos y conjunto de instrucciones sobre dichos datos que se deben ejecutar en secuencia.
- **Programa concurrente**: conjunto de programas secuenciales ordinarios que se pueden ejecutar *lógicamente* en paralelo.
- **Proceso**: Ejecución de un programa secuencial.
- **Concurrencia**: Describe el potencial para ejecución paralela, es decir, el solapamiento real o virtual de varias actividades en el tiempo.
- **Programación concurrente**: Conjunto de notaciones y técnicas de programación usadas para expresar paralelismo potencial y resolver problemas de sincreonización y comunicación. La programación concurrente es independiente de la implementación del paralelismo. Es una abstracción.
- **Programación paralela**: Su principal objetivo es acelerar la resolución d eproblemas concretos mediante el aprovechamiento de la capacidad de procesamiento en paralelo del hardware disponible.
- **Programación distribuida**: Su principal objetivo es hacer que varios componentes software localizados en diferentes ordenadores trabajen juntos.
- **Programación de tiempo real**: Se centra en al programación de sistemas que están funcionando continuamente, recibiendo entradas y enviando salidas a/desde componentes hardware en los que se trabaja con restricciones muy estrictas en cuanto a la respuesta temporal.

1.1.2 2. Modelo abstracto y consideraciones sobre el hardware

2.1 Consideraciones sobre el hardware La concurrencia puede implementarse en distintos tipos de hardware.

En monoprocesadores dan lugar a un paralelismo virtual, para gestionar el reparto de ciclos de CPU.

En multiprocesadores, tanto de memoria compartida o en sistemas distribuidos, mejora la distribución de los trabajos para acelerar los tiempos de cálculo. #### 2.2 Modelo abstracto de concurrencia

Sentencia atómica Una sentencia o instrucción de un proceso en un programa concurrente es atómica (indivisble) si siempre se ejecuta de principio a fin sin verse afectada por otras sentencias en ejecución de otros procesos del programa.

- No se verá afectada cuando el funcionamiento de dicha instrucción no dependa nunca de como se estén ejecutando otras instrucciones.
- El funcionamiento de una instrucción se define por su efecto en el estado de ejecución del programa justo cuando acaba.
- El estado de ejecución está formado por los valores de las variables y los registros de todos los procesos.

Algunos ejemplos de intrucciones atómicas son:

- Leer una celda de memoria y cargar su valor en ese momento en un registro del procesador.
- Incrementar el valor de un registro (u otras operaciones aritméticas entre registros).
- Escribir el valor de un regristro en una celda de memoria.

La mayoría de sentencias en lenguajes de alto nivel son típicamente no atómicas. Por ejemplo, la sentencia
$x = x+1$
consta en realidad de una secuencia de 3 sentencias:

1. Leer el valor de x y cargarlo en un registro r del procesador.
2. Incrementar en una unidad el valor almacenado en el regristro r.
3. Escribir el valor del registro r en la variable x.

El resultado depende de que haya o no otras sentencias ejecutándose a la vez y escribiendo simultáneamente sobre la variable x. Hay indeterminación (no se puede predecir el estado final a partir del inicial).

Si tenemos un programa C compuesto por dos procesos P_A y P_B que se ejecutan a la vez, las ejecución sentencias atómicas de cada uno de estos procesos pueden darse en múltiples órdenes (interfoliaciones). Las sentencias atómicas se ordenan en función del instante en que acaban. El modelo basado en el estudio de las posibles secuencias de ejecución entrelazadas de los procesos constituye una abstracción: - Se consideran exclusivamente caracterísitcas relevantes que determinan el resultado del cálculo. - Esto permite simplificar el análisis y diseño de los programas concurrentes.

Se ignoran los detalles no relevantes para el resultado, como por ejemplo:

- Las áreas de memoria asignadas a los procesos.
- Los registros particulares que están usando.
- El costo de los cambios de contexto entre procesos.
- La política del SO relativa a la asignación de CPU.
- Las diferencias entre entornos multiprocesador o monoprocesador.

El entrelazamiento preserva la consistencia El resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución. Supongamos que un programa *P* se compone de dos instrucciones atómicas, *I_0* e *I_1* que se ejecutan concurrentemente. Entonces: - Si *I_0* e *I_1* no acceden a la misma celda de memoria o registro, el orden de ejecución no afecta al resultado final. - Si *I_0* carga en la posición *M* de memoria un dato 1, e *I_1* carga un dato 2, entonces al final de la ejecución se dará $M = 1$ o $M = 2$, pero nunca $M = 3$.

Progreso finito No se puede hacer una suposición acerca de las velocidades absolutas o relativas de ejecución de los procesos, salvo que son mayores que 0. Un programa concurrente se entiende en basesa a sus componentes (procesos) y sus interacciones, sin tener en cuenta el entorno de ejecución.

Si se cumple la hipótesis, la velocidad de ejecución de cada proceso será no nula, lo cual tiene dos consecuencias. Desde el punto de vista global, durante la ejecución del programa concurrente, en cualquier momento existirá al menos 1 proceso preparado. Desde el punto de vista local, cuando un proceso concreto de un programa concurrente comienza la ejecución de una sentencia, completará la ejecución de la sentencia en un intervalo de tiempo finito.

Estado e historia de ejecución de un programa concurrente El estado de un programa concurrente son los valores de las variables del programa en un momento dado. Incluyen variables declaradas explícitamente y variables con información de estado oculta. Un programa concurrente comienza su ejecución en un estado inicial y los procesos van modificando el estado conforme se van ejecutando sus sentencias atómicas.

La historia o traza de un programa concurrente son secuencias de estados producidas por una secuencia concreta de interfoliación.

Grafo de sincronización El *grafo de sincronización* es un grafo dirigido acíclico donde cada nodo representa una secuencia de sentencias del programa.

Dadas dos actividaddes *A* y *B*, una arista desde *A* hacia *B* significa que *B* no puede comenzar su ejecución hasta que *A* haya finalizado.

El grafo de sincronización muestra las restricciones de precedencia que determinan cuándo una actividad puede empezar en un programa.

Definición estática de procesos El número de procesos (arbitrario) y el código que ejecutan no cambian entre ejecuciones. Cada proceso se asocia con su identificador y su código mediante la palabra clave process.

El programa acaba cuando acaban todos los procesos. Las variables compartidas se inicializan antes de comenzar ningún proceso.

Sew pueden usar definiciones estáticas de grupos de procesos similares que solo se diferencian en el valor de una constante (vectores de procesos).

Creación de procesos no estructurada con fork-join **fork**: sentencia que especifica que la rutina nombrada puede comenzar su ejecución al mismo tiempo que comienza la sentencia siguiente (bifurcación).

join: sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente (unión).

Creación de procesos no estructurada con cobegin-coend Las sentencias en un bloque delimitado por cobegin-coend comienzan su ejecución todas ellas a la vez.

En el coend es espera a que se terminen todas las sentencias. Hace explícito qué rutinas van a ejecutarse concurrentemente.

1.1.3 3. Exclusión mutua y sincronización

3.1 Concepto de exclusión mutua Según el modelo abstracto, los procesos concurrentes ejecutan sus instrucciones atómicas de forma que, en principio, el entremezclado en el tiempo es arbitrario. Sin embargo, en un conjunto de procesos que no son independientes entre sí (cooperativos) algunas de las posibles formas de combinar las secuencias no son válidas.

En general, se dice que hay una condición de sincronización cuando esto ocurre, es decir, que hay alguna restricción sobre el orden en el que se pueden mezclar las instrucciones de distintos procesos.

Un caso particular es la exclusión mutua, secuencias finitas de intrucciones que deben ejecutarse de principio a fin por un único proceso, sin que a la vez otro proceso las esté ejecutando también.

La restricción se refiere a una o varias secuencias de instrucciones consecutivas que aparecen en el texto de uno o varios procesos. Al conjunto de dichas secuencias de instrucciones se le denomina sección crítica (SC). Ocurre exclusión mutua (EM) cuando los procesos solo funcionan correctamente si, en cada instante de tiempo, hay como mucho uno de ellos ejecutando cualquier instrucción de la serie crítica.

El solapamiento de las instrucciones debe ser tal que cada secuencia de instrucciones de la SC se ejecuta como mucho por un proceso de principio a fin, sin que durante ese tiempo otros procesos ejecuten ninguna de estas instrucciones ni otras de la misma SC.

Un ejemplo típico de EM ocurre en procesos con memoria compartida que acceden para leer y modificar variables o estructuras de datos comunes usando operaciones no atómicas.

3.2 Condición de sincronización En general, en un programa concurrente compuesto de varios procesos, una condición de sincronización establece que no son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos. Eso ocurre típicamente cuando, en un punto concre-

to de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global.

Ejemplo: productor-consumidor

1.1.4 4. Propiedades de los sistemas concurrentes

Una propiedad de un programa concurrente es un atributo del programa que es cierto para todas las posibles secuencias de interfoliación. Hay de dos tipos

Propiedades de seguridad Son condiciones que deben cumplirse en cada instante (nunca sucederá algo malo). Requeridas en especificaciones estáticas del programa. Son fáciles de demostrar y para cumplirlas se suelen restringir las posibles interfoliaciones.

Ejemplos:

- Exclusión mutua: 2 procesos nunca entrelazan ciertas subsecuencias de operaciones.
- Ausencia de interbloqueo: nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá.
- Propiedad de seguridad en el productor-consumidor: el consumidor debe consumir todos los datos producidos por el productor en el orden en el que se van produciendo.

Propiedades de vivacidad Son propiedades que deben cumplirse eventualmente (realmente sucede algo bueno). So propiedades dinámicas, más difíciles de probar.

Ejemplos:

- Ausencia de inanición: un proceso o grupo de procesos no puede ser indefinidamente pospuesto. En algún momento, podrá avanzar.
- Equidad: tipo particular de propiedad de vivacidad. Un proceso que desee progresar debe hacerlo con justicia relativa con respecto a los demás. Más ligado a la implementación y a veces incumplida: existen distintos grados.

1.1.5 5. Verificación de programas concurrentes

5.1 Introducción £Cómo demostrar que un programa cumple una determinada propiedad?

- Posibilidad: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad. Problema: sólo permite considerar un número limitado de historias (interfoliaciones) de ejecución y no demuestra que no existan casos indeseables.

- Enfoque operacional: Análisis exhaustivo de casos. Se chequea la corrección de todas las posibles historias. Problema: su utilidad está muy limitada cuando se aplica a programas concurrentes complejos ya que el número de interfoliaciones crece exponencialmente con el número de instrucciones de los procesos.

5.2 Enfoque axiomático Se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia. Se usan formas lógicas (asertos) para caracterizar un conjunto de estados. Las sentencias atómicas actúan como transformadores de predicados (asertos). Los teoremas en la la lógica tienen la forma: *{P} S {Q}* Si la ejecución de la sentencia *S* empieza en algún estado en el que es verdadero el predicado *P* (precondición), entonces el predicado *Q* (poscondición) será verdadero en el estado resultante.

El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas del programa.

Invariante global es un predicado que referencia variables globales siendo cierto en el estado inicial de cada proceso y manteniéndose cierto ante cualquier asignación dentro de los procesos.

Ejemplo para productor consumidor:
$consumidos <= producidos <= consumidos + 1$

1.2 Tema 2. Sincronización en memoria compartida

1.2.1 1. Introducción a la sincronización en memoria compartida

En este tema estudiaremos soluciones para exclusión mutua y sincronización basadas en el uso de memoria compartida entre los procesos involucrados. Las soluciones pueden ser:

- **Bajo nivel con espera ocupada**. Basadas en programas que contienen explícitamente instrucciones de bajo nivel para la lectura y escritura directamente a la memoria compartida y bucles para realizar las esperas.
- **Alto nivel**. Partiendo de las anteriores, se diseña una capa software por encima que ofrece un interfaz a las aplicaciones. La sincronización se consigue bloqueando un proceso cuando deba esperar.

Soluciones a bajo nivel con espera ocupada Cuando un proceso debe esperar a que ocurra un evento o sea cierta determinada condición, entra en un bucle indefinido el cual continuamente comprueba si la situación ya se da o no (espera ocupada). Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones software**. Operaciones sencillas de lectura y escritura de datos simples en la memoria compartida.

- **Soluciones hardware (cerrojos)**. Basadas en la existencia de instrucciones máquina específicas dentro del repertorio de instrucciones de los procesadores involucrados.

Las soluciones de bajo nivel con espera ocupada se prestan a errores, producen algoritmos complicados y tienen un impacto negativo en la eficiencia. En las soluciones de alto nivel se ofrecen interfaces de acceso a estructuras de datos y además se usa bloqueo de procesos en lugar de espera ocupada.

Entre estas soluciones de alto nivel se encuentran **semáforos**, **regiones críticas condicionales** y **monitores**. Todas estas las iremos detallando a continuación.

1.2.2 2. Semáforos para sincronización

2.1. Introducción Los semáforos constituyen un mecanismo que soluciona o aminora los problemas de las soluciones de bajo nivel, y tienen un ámbito más amplio:

- No se usa espera ocupada, sino bloqueo de procesos (uso más eficiente de CPU).
- Resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos.
- Se pueden usar para resolver problemas de sincronización.
- El mecanismo se implmenta mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos. Esto aumenta la seguridad y la simplicidad.

Los semáforos exigen que los procesos en espera no ocupen la CPU, esto implica que:

- Un proceso en ejecución debe poder solicitar quedarse bloqueado.
- Un proceso bloqueado no puede ejecutar instrucciones en la CPU.
- Un proceso en ejecución debe poder solicitar que se desbloquee (se reanude) algún otro proceso bloqueado.
- Deben poder existir simultáneamente varios conjuntos de procesos bloqueados.
- Cada petición de bloque o desbloqueo se debe referir a alguno de estos conjuntos.

Todo esto requiere el uso de servicios externos proporcionados por el sistema operativo o la librería de hebras.

2.2. Estructura de un semáforo Un semáforo es una instancia de una estructura de datos que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (de estos procesos decimos que están esperando al semáforo).
- Un valor natural (entero no negativo) al que llamaremos por simplicidad valor del semáforo.

Estas estructuras de datos residen en memoria compartida. Al inicio de un programa que los usa debe poder inicializarse cada semáforo:

- El conjunto de procesos asociados estará vacío.
- Se deberá indicar un valor inicial del semáforo.

2.3. Operaciones sobre los semáforos Además de la inicializacíon, solo hay dos operaciones básicas que se pueden realizar sobre una variable del tipo semáforo (que llamamos s):

- **sem_wait(s)**
  1. Si el valor de s es 0, bloquear el proceso, que será reanudado después en un instante en el que el valor ya es 1.
  2. Decrementar el valor del semáforo en una unidad.
- **sem_signal(s)**
  1. Incrementar el valor de s en una unidad.
  2. Si hay procesos esperando en s, reanudar uno de ellos.

Este diseño implica que el valor del semáforo nunca es negativo, ya que antes de decrementar se espera a que sea 1. Además, solo puede haber procesos esperando cuando el valor es 0.

Dado un semáforo s, en un instante de tiempo cualquiera t su valor será el valor iniciar más el número de llamadas sem_signal completadas menos el número de llamadas sem_wait completadas. Este valor nunca es negativo.

2.4. Uso de semáforos. Patrones sencillos Veremos la solución de tres problemas mediante el uso de semáforos.

Espera única Para introducir una espera, usamos un semáforo cuyo valor es 1 solo cuando x tiene un valor ya escrito pero aun está pendiente de leer. La inicialización es, por tanto, a 0.

Exclusión mutua Los semáforos se pueden usar para EM usando un semaforo inicializado a 1, haciendo wait antes de la sección crítica y signal después.

Productor consumidor La solución es parecida a espera única, pero en un bucle infinito. Se utiliza un nuevo semáforo puede_escribir inicializado a 1.

2.5 Limitaciones de los semáforos Los semáforos resuelven de una forma eficniente y sencilla el problema de exclusión mutua y problemas sencillos de sincronización, sin embargo:

- Los problemas de sincronización más complejos se resuelven de forma más compleja
- Al igual que los cerrojos, programas erróneos o malintencionados pueden provocar que haya procesos bloqueados indefinidamente o en estado incorrecto.

En la siguiente sección veremos una solución a más alto nivel sin estas limitaciones.

1.2.3 3. Monitores como mecanismos de alto nivel

3.1. Introducción. Definición de monitor C.A.R. Hoare en 1947 introduce el concepto de monitor. Es un mecanismo de alto nivel que perminte definir objetos abstractos compartidos, que incluyen:

- Una colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos.
- Un conjunto de procedimientos para manipular el recurso: afectan a las variables encapsuladas.

Ambos conjuntos de elementos permiten al programador invocar los procedimientos de forma que en ellos:

- Se garantice el acceso en exclusión mutua a las variables encapsuladas.
- Se implemente la sincronización requerida por el problema mediante esperas bloqueadas.

Un monitor es un recurso compartido que se usa como un objeto al que se accede concurrentemente.

- El usuario solo puede acceder al recurso mediante un conjunto de operaciones.
- El usuario ignora la/s variable/s que representan al recurso y la implementación de las operaciones asociadas.
- La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por definición.
- La implementación del monitor garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor.

Las propiedades descritas de los monitores los hacen preferibles respecto de los semáforos, dado que el uso de monitores facilita el diseño e implementación de programas libres de errores.

- Las variables están protegidas: solo pueden leerse o modificarse desde el código del monitor.
- La exclusión mutua está garantizada: el programador no tiene que usar mecanismos explícitos de exclusión mutua en el acceso a las variables compartidas.

Componentes de un monitor.

- Las variables permanentes son el estado interno del monitor. Sólo pueden ser accedidas dentro del monitor. Permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor.
- Procedimientos. Modifican el estado interno. Pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento. Algunos constituyen una interfaz externa del monitor y podrán ser llamados por los procesos que comparten el recurso.
- Código de inicialización. Fija el estado interno inicial (opcional). Se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor.

3.2. Funcionamiento de los monitores *Comunicación monitor-mundo exterior*: Cuando un proceso necesita operar sobre un recurso compartido controlado por un monitor deberá realizar una llamada a uno de los procedimientos exportados por el monitor usando los parámetros actuales apropiados. Mientras el proceso está ejecutando algún proceidmiento del monitor decimos que el proceso está dentro del monitor.

Exclusión mutua: Si un proceso P está dentro de un monitor, cualquier otro proceso Q que llame a un procedimiento de ese monitor deberá esperar hasta que P salga del mismo. Esta política de acceso asegura que las variables permanentes nunca son accedidas concurrentemente. El acceso exclusivo entre los procedimientos del monitor debe estar garantizado en la implementación de los monitores.

Monitores son objetos pasivos: Después de ejecutarse el código de inicialización, un monitor es un objeto pasivo y el código de sus procedimientos solo se ejecuta cuando estos son invocados por los procesos.

Instancias de monitores: En algunos casos es conveniente crear múltiples instancias independientes de un monitor. Cada instancia tiene sus variables permanentes propias. La exclusión mutua ocurre en cada instancia por separado. Esto facilita mucho escribir código reentrante.

Cola del monitor para exclusión mutua El control de la exclusión mutua se basa en la existencia de la cola del monitor: - Si un proceso está dentro del monitor y otro intenta ejecutar un procedimiento del monitor, este último proceso queda bloqueado y se inserta en la cola del monitor.

- Cuando un proceso abandona el monitor, se desbloquea un proceso de la cola, que ya puede entrar al monitor.
- Si la cola del monitor está vacía, el monitor está libre y el primer proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor.
- Para garantizar la vivacidad del sistema, la planificación de la cola debe seguir una política FIFO.

3.3. Sincronización de monitores Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición. Para cada condición distinta que los procesos pueden eventualmente tener que esperar en un monitor, se debe declarar una variable permanente de tipo condition. A esas variables las llamamos señales o variables de condición.

Cada variable de condición tiene asociada una lista o cola inicialmente vacía de procesos en espera hasta que la condición se haga cierta.

Para una cualquiera de estas variables, un proceso puede invocar las operaciones wait y signal (esperar a la condición y señalar que la condición ocurre).

Además, cond.queue() devuelve true si hay algún proceso esperando en la cola cond, y false en caso contrario.

Dado que los procesos pueden estar dentro del monitor, pero bloqueados:

- Cuando un proceso llama a wait y queda bloqueado, se debe liberar la exclusión mutua del monitor, si no se hiciese, se produciría un interbloqueo con seguridad.
- Cuando un proceso es reactivado después de una espera, adquiere de nuevo la exlusión mutua antes de ejecutar la sentencia siguiente a wait.
- Más de un proceso podrá estar dentro de monitor, aunque solo uno de ellos estará ejecutándose, el resto estarán bloqueados en variables de condición.

3.5. Colas de condición con prioridad Por defecto se usan colas de espera FIFO. Sin embargo a veces resulta útil disponer de un mayor control sobre la estrategia de planificación, dando la prioridad del proceso en espera como un parámentro entero de wait. cond.signal() reanudará un proceso que especificó el valor mínimo de prioridad de entre todos los que esperan (si hay más de uno con la prioridad mínima se usa política FIFO). Se deben evitar riesgos de inanición. No tiene ningún efecto sobre la lógica del programa: el funcionamiento es similar con y sin colas de prioridad. Sólo mejoran las caracterísiticas dependientes del tiempo.

3.7. Semántica de las señales de los monitores Cuando un proceso hace signal en una cola no vacía, se denomina proceso señalador. El proceso que espera en la cola y se reactiva se denomina señalado. Suponemos que hay un código restante

del monitor tras el wait y el signal. Inmediatamente después de señalar, no es posible que ambos continuen la ejecución de su código restante, ya que no se cumpliría la exclusión mutua del monitor. Uno de los dos puede inmediatamente ejecutar su código restante, pero entonces el otro no puede hacerlo. Se denomina semántica de señales a la política que establece la forma concreta en que se recuelve el conflicto tras hacerse un signal en una cola no vacía.

- El proceso señalador continúa su ejecución tras el signal. El señalado espera bloqueado hasta que puede adquirir la EM de nuevo (SC: señalar y continuar).
- El proceso señalado se reactiva inmediatamente. El señalador:
  - abandona el monitor tras hacer el signal sin ejecutar el código después de dicho signal (SS: señalar y salir).
  - queda bloqueado a la espera en:

    la cola del monitor, junto con otros posibles procesos que quieren comenzar a ejecutar código del monitor (SE: señalar y esperar)
  - una cola específica para esto, con mayor priotidad que esos procesos (SU: señalar y espera urgente)

Señalar y continuar El proceso señalado continúa su ejecución dentro del monitor después del signal. El proceso señalado abandona la cola de condición y espera en la cola del monitor para readquirir la EM.

Tanto el señalador como otros procesos pueden hacer falsa la condición después de que el señalado abandone la cola de condición. Por tanto, en el proceso señalado no se puede garantizar que la condición asociada a cond es cierta al terminar el cond.wait(), y por lo que es necesario volver a comprobarla entonces. Esta semántica obliga a programar la operación wait en un bucle.

Señalar y salir El proceso señalador sale del monitor después de ejecutar cond.signal(). Si hay código tras el signal, no se ejecuta. El proceso señalado reanuda inmediatamente la ejecución del código del monitor.

En ese caso, la operación signal conlleva liberar al proceso señalado y terminar el procedimiento del monitor que estaba ejecutando el proceso señalador. Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó. Esta semántica condiciona el estilo de programación ya que obliga a colocar la operación signal como última instrucción de los procedimientos de monitor que la usen.

Señalar y esperar El proceso señalador se bloquea en la cola del monitor justo después de ejecutar signal. El proceso señalado entra de forma inmediata en el monitor.

Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó. El proceso señalador entra en la cola de procesos del monitor, por lo que está al mismo nivel que el resto de procesos que compiten por la exclusión mutua del monitor. Puede considerarse una semántica injusta respecto al proceso señalador ya que dicho proceso ya había tenido acceso al monitor por lo que debería tener prioridad sobre el resto de procesos que compiten.

Señalar y espera urgente Es similar a SE, pero se intenta corregir el problema de la falta de equitatividad indicado.

El proceso señalador se bloquea justo después de ejecutar signal. El proceso señalado entra de forma inmediata en el monitor. Está asegurado el estado que perminte al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó. El proceso señalador entra en una nueva cola de procesos que espera para acceder al monitor, que podemos llamar cola de procesos urgentes. Los procesos de la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor. Es la semántica que se supone en los ejemplos vistos.

Análisis comparativo de las diferentes semánticas

- Potencia expresiva: todas las semánticas son capaces de resolver los mismos problemas.
- Facilidad de uso: La semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos.
- Eficiencia
  - SE y SU resultan ineficientes cuando no hay código tras signal, ya que en ese caso implican que el señalador emplea tiempo en desbloquearse y después reactivarse, pero justo a continuación abandona el monitor sin hacer nada.
  - La semánctica SC es un poco ineficiente al obligar a usar un bucle para cada instrucción signal.

3.9. Verificación de programas con monitores La verificación de la corrección de un programa concurrente con monitores requiere:

- Probar la corrección de cada monitor
- Probar la corrección de cada proceso de forma aislada
- Probar la corrección de la ejecución concurrente de los procesos implicados

El programador no debe conocer a priori la interfoliación concreta de llamadas a los procedimientos del monitor. El enfoque de verificación que vamos a seguir utiliza una invariante de monitor:

- Establece una relación constante entre valores permitidos de las variables del monitor y/o las interfoliaciones que ocurren.
- Debe ser cierto siempre, excepto cuando un proceso está ejecutando código en EM (cambiando el estado del monitor).

El invariante del monitor será una función lógica que se evalua en cada estado en términos de los valores de las variables permanentes en ese estado y/o las posibles interfoliaciones que han ocurrido hasta llegar a ese estado. El invariante debe ser cierto siempre que no haya un proceso ejecutándose en el monitor. Debe ser cierto por tanto:

- En el estado inicial, después de la inicialización de las variables permanentes.
- Antes y después de cada llamada a un procedimiento del monitor.
- Antes y después de cada operación wait.
- Antes y depsués de cada operación signal.

Justo antes de signal sobre una variable de condición, además, debe ser cierta la condición lógica asociada a dicha variable.

1.2.4 4. Soluciones software con espera ocupada para la EM

En esta sección veremos diversas soluciones para lograr la exclusión mutua en una sección crítica usando variables compartidas entre los procesos o hebras involucrados.

- Estos algoritmos usan dichas variables para hacer espera ocupada cuando sea necesario en el protocolo de entrada.
- Los algoritmos que resuelven este problema no son triviales, y menos para más de dos procesos. En la actualidad se conocen distintas soluciones con distintas propiedades, veremos el algoritmo de Dekker (para 2 procesos) y el de Peterson (para 2 y para un número arbitrario de procesos).
- Previamente a esos algoritmos, veremos la estructura de los procesos con secciones críticas y las propiedades que deben cumplir los algoritmos.

4.1. Estructura de los procesos con secciones críticas Para analizar las soluciones a EM asumimos que un proceso que incluya un bloque considerado como secicón crítica (SC) tendrá dicho bloque estructurado en tres etapas:

1. Protocolo de entrada (PE): una serie de instrucciones que incluyen posiblemente espera, en los casos en los que no se pueda conceder acceso a la sección crítica.
2. Sección crítica (SC): instrucciones que solo pueden ser ejecutadas por un proceso como mucho.

3. Protocolo de salida (PS): instrucciones que permiten que otros procesos puedan conocer que este proceso ha terminado la sección crítica.

Todas las sentencias que no forman parte de ninguna de estas tres etapas se denominan resto de sentencias (RS).

Para que se puedan implementar soluciones correctas al problema de EM, es necesario suponer que los procesos siempre terminan una sección crítica y emplean un intervalo de tiempo finito desde que comienzan hasta que la terminan. Durante el tiempo que un proceso se encuentra en una sección crítica, nunca:

- Finaliza o aborta.
- Es finalizado o abortado externamiente.
- Entra en un bucle infinito.
- Es bloqueado o suspendido indefinidamente de forma externamiente.

En general, es deseable que el tiempo empleado en las secciones críticas sea el menor posible

4.1. Propiedades para la exclusión mutua Para que un algoritmo para EM sea correcto, se deben cumplir cada una de estas tres propiedades mínimas:

1. Exclusión mutua
2. Progreso
3. Espera limitada

Además, hay propiedades deseables adicionales que también deben complirse:

4. Eficiencia
5. Equidad

Si bien consideramos correcto un algoritmo que no sea muy eficiente o para el que no pueda demostrarse claramente la equidad.

Propiedad de exclusión mutua Es la propiedad fundamental para el problema de la sección crítica. Establece que en cada instante de tiempo, y para cada sección crítica existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha región crítica.

En esta sección veremos soluciones de memoria compartida que permiten un único proceso en una sección crítica. Si bien esta es la propiedad fundamental, no puede conseguirse de cualquier forma, y para ello se establecen las otras dos condiciones mínimas que veremos a continuación.

Propiedad de progreso Consideremos una SC en un instante en el cual no hay ningún proceso ejecutándola, pero sí hay procesos en el PE compitiendo por entrar a la SC. La propiedad de progreso establece:

Un algortimo de EM debe de estar diseñado de forma tal que:

1. Después de un intervalo de tiempo finito desde que ingresó en el primero proceso al PE, uno de los procesos en el mismo podrá acceder a la SC.
2. La sección del proceso anterior es completamente independiente del comportamiento de los procesos que durante todo ese intervalo no han estado en SC ni han intentado acceder.

Cuando la condición (1) no se da, se dice que ocurre un interbloqueo, ya que todos los procesos en el PE quedan en espera ocupada indefinidamente sin que ninguno pueda avanzar.

Espera limitada Supongamos que un proceso emplea un intervalo de tiempo en el PE intentando acceder a un SC. Durante ese intervalo de tiempo, cualquier otro proceso activo puede entrar un número arbitrario de veces n a ese mismo PE y lograr acceso a la SC (incluyendo la posibilidad de que $n = 0$). La propiedad de espera limitada establece que:

Un algoritmo de exclusión mutua debe estar diseñado de forma que n nunca será superior a un valor máximo determinado.

Esto implica que las esperas en el PE siempre serán finitas.

Propiedades deseables: eficiencia y equidad Las propiedades deseables son estas:

- **Eficiencia**. Los protocolos de entrara y salida deben emplear poco tiempo de procesamiento (excluyendo las esperas ocupadas del PE), y las variables compartidas deben usar poca cantidad de memoria.
- **Equidad**: En los casos en que haya varios procesos compitiendo por acceder a una SC (de forma repetida en el tiempo), no debería existir la posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros.

4.3. Refinamiento sucesivo de Dijsktra El refinamiento sucesivo de Dijsktra hace referencia a una serie de algoritmos que intentan resolver el problema de la exclusión mutua.

- Se comienza desde una versión simple, incorrecta, y se hacen sucesivas mejoras para intentar cumplir las tres propiedades.
- La versión final correcta se denomina algoritmo de Dekker.
- Por simplicidad, veremos algoritmos para dos procesos únicamente.
- Se asume que hay dos procesos, P0 y P1, cada uno de ellos ejecuta un bucle infinito conteniendo: protocolo de entrada, sección crítica, protocolo de salida y otras sentencias del proceso.

4.4. Algoritmo de Dekker El algortimo de Dekker es correcto y se puede interpretar como el final del refinamiento sucesivo de Dijkstra.

- Al igual que en la versión 5, cada proceso incorpora una espera de cortesía durante la cual le cede al otro la posiblidad de entrar al SC cuando ambos coinciden en PE.
- Para evitar interbloqueos, la espera de cortesía solo la realiza uno de los dos procesos, de forma alterna, mediante una variable turno.
- La variable de turno permite tambíen saber cuando acabar la espera de cortesía, que se implementa mediante un bucle.

4.5. Algoritmo de Peterson Este es otro algoritmo correcto para EM, más simple que el de Dekker.

- Al igual que el de Dekker, usa dos variables lógicas que expresan la presencia de cada proceso en el PE o la SC, más una variable de turno que permite romper el interbloqueo en caso de acceso simultáneo al PE.
- La asignación a la variable de turno se hace al inicio del PE en lugar de en el PS, con lo cual, en caso de acceso simultáneo al PE, el segundo proceso en ejecutar la asignación (atómica) al turno da preferencia al otro (el primero en llegar).
- A diferencia del algoritmo de Dekker, el PE no usa dos bucles anidados, sino que unifica ambos en uno solo.

1.2.5 5. Soluciones hardware con espera ocupada (cerrojos) para EM

5.1. Introducción Los cerrojos constituyen una solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con memoria compartida para solucionar el problema de la exclusión mutua.

- La espera ocupada constituye un bucle que se ejecuta hasta que ningún otro proceso esté ejecutando intrucciones de sección crítica.
- Existe un valor lógico en una posición de memoria compartida (llamado cerrojo) que indica si algún proceso está en sección crítica.
- En el protocolo de salida se actualiza el cerrojo de forma que se refleje que la SC ha quedado libre.

5.2. La instrucción TestAndSet Es una instrucción máquina disponible en el repertorio de algunos procesadores. Admite como argumento la dirección de memoria de la variable lógica que actúa como cerrojo. Se invoca como una función desde LLPP de alto nivel, y ejecuta estas acciones:

1. Lee el valor anterior del cerrojo
2. Pone el cerrojo a true

3. Devuelve el valor anterior del cerrojo

Durante su ejecución, ninguna otra instrucción ejecutada por otro proceso puede leer ni escribir la variable lógica: por tanto, se ejecuta de forma atómica.

5.3 Desventajas de los cerrojos Los cerrojos constituyen una solución válida para EM que consume poca memoria y es eficiente en tiempo, sin embargo:

- Las esperas ocupadas consume tiempo de CPU que podría dedicarse a otros procesos para hacer trabajo útil.
- Se puede acceder directamente a los cerrojos y por tanto un programa erróneo o escrito malintencionadamente puede poner un cerrojo en un estado incorrecto, pudiendo dejar a otros procesos indefinidamente en espera ocupada.
- No se cumplen condiciones de equidad.

5.4. Uso de los cerrojos Las desventajas indicadas hacen que el uso de cerrojos sea restringido, en el sentido de que:

- Por seguridad, normalmente solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares.
- Para evitar la pérdida de eficiencia que supone la espera ocupada, se usan solo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto.

1.3 Tema 3. Sistemas basados en paso de mensajes

1.3.1 1. Mecanismos básicos en sistemas basados en paso de mensajes

1.1 Introducción Sistemas distribuidos: conjunto de procesos en uno o varios ordenadores que no comparten memoria, pero se transmiten datos a través de una red. Facilita la distribución de datos y recursos. Soluciona el problema de la escalabilidad y el elevado coste. Presenta mayor dificultad de programación ya que no hay direcciones de memoria comunes y mecanismos como los monitores son inviables.

A las acciones que se realizan en memoria común (asignación, estructuración) se le añaden dos nuevas, envío y recepción, que afectan al entorno externo. Permiten comunicar procesos que se ejecutan en paralelo.

1.2 Vista lógica de la arquitectura y modelo de ejecución

Vista lógica de la arquitectura Existen N procesos, cada uno con su espacio de direcciones propio (memoria). Los procesos se comunican mediante envío y recepción de mensajes. EN un mismo procesador pueden residir físicamente varios procesos. Por motivos de eficiencia, frecuentemente se adopta la política de alojar un único proceso en cada procesador disponible. Cada iteración requiere cooperación entre 2 procesos: el propietario de los datos (emisor) debe intervenir aunque no haya conexión lógica con el evento tratado en el receptor.

Estructura de un programa de paso de mensajes. SPMD Diseñar un código diferente para cada proceso puede ser complejo. Una solución es el estilo SPMD (Single Program Multiple Data):

- Todos los procesos ejecutan el mismo código fuente.
- Cada proceso puede procesar datos distintos y/o ejecutar flujos de control distintos.

Estructura de un programa de paso de mensajes. MPMD Otra opción es usar el estilo MPMD (Multiple Program Multiple Data):

- Cada proceso ejecuta el mismo o diferentes programas de un conjunto de ficheros ejecutables.
- Los diferentes procesos pueden usar datos diferentes.

1.3 Primitivas básicas de paso de mensajes El paso de un mensaje entre dos procesos constituye una transferencia de una secuencia finita de bytes. 1. Se leen de una viariable del proceso emisor (var_orig). 2. Se transfieren a través de alguna red de interconexión. 3. Se escriben en una ariable del proceso receptor (var_dest). Implica sincronización: los bytes acaban de recibirse después de iniciado el envío. Ambas variables son del mismo tipo. EL efecto neto final es equivalente a una hipotética asignación (var_dest := var_orig).

Primitivas básicas de paso de mensajes EL proceso emisor realiza el envío invocando a send, y el proceso receptor realiza la recepción invocando a receive. La sintaxis es:

- send (variable origen, identificador_proceso_destino)
- receive (variable destino, identificador_proceso_origen)

Cada proceso nombra explícitamente al otro, indicando su nombre de proceso como identificador.

£Cómo identifica el emisor al receptor del mensaje y viceversa? Existen dos posibilidades:

- Denominación directa estática

  - El emisor identifica explícitamente al receptor y viceversa.
  - Para la identificación de usan identificadores de procesos. Cada identificador es un valor (típicamente un entero) biunívocamente asociado a un proceso en tiempo de compilación.

- Denominación indirecta
  - Los mensajes se depositan en almacenes intermedios que son accesibles desde todos los procesos.
  - El emisor nombra un buzón al que envía. El receptor nombra un buzón desde el que quiere recibir.

Denominación directa estática Ventajas - No hay retardo para establecer la identificación (los símbolos P0 Y P1 se traducen en dos enteros en una implementación).

Inconvenientes - Cambios en la identificación requieren recompilar el código. - Sólo permite comucicación en pares.

Denominación directa con identificación asimétrica Existen otras posibilidades llamadas esquemas asimétricos: el emisor identifica al receptor, pero el receptor no indica un emisor específico.

El receptor inicia una recepción, pero indica que acepta recibir el mensaje de cualqueir otro posible proceso emisor. Una vez recibido el mensaje, el recoptor puede conocer que proceso ha sido emisor, de dos formas, ya que el identificador del emisor puede formar parte de los metadatos del mensaje o puede ser un parámetro de salida de receive. También es posible especificar que el emisor debe pertenecer a un subconjunto de todos los posibles emisores (se deben definir previamente los subconjuntos de forma coordinada).

Denominación indirecta En la denominación indirecta, el emisor y el receptor identifican un buzón o canal intermedio a través del cual se transmiten los mensajes. El uso de buzanes da mayor flexibilidad ya que permite comunicaicón entre múltiples receptores y emisores.

Existen tres tipos de buzones: canales (uno a uno), puertos (muchos a uno) y buzones generales (muchos a muchos).

Un mensaje enviado a un buzón general permanece en dicho buzón hasta que hay sido leído por todos los procesos receptores potenciales.

Declaración estática vs dinámica Los identificadores de proceso suelen ser valores enteros, cada uno de ellos está biunívocamente asociado a un proceso del programa concurrente. Se pueden gestionar: - Estáticamente: en el código fuente se asocian explícitamente los números enteros a los procesos: - Ventaja: es muy eficiente en tiempo. - Inconveniente: cambios en la estructura del programa requieren adaptar el código fuente y recompilarlo.

- Dinámicamente: el identificador numérico de cada proceso se calcula en tiempo de ejecución cuando es necesario:
  - Desventajas: es menos eficiente en tiempo, más complejo.
  - Ventaja: el código puede seguir siendo válido aunque cambie el número de procesos de cada tipo.

Instantes críticos en el lado del emisor Para poder transmitir un mensaje el sistema de paso de mensajes (SPM), en el lado del emisor, debe dar estos pasos:

1. Fin del registro de la soliucitud de envío (SE): después de iniciada la llamada send, el SPM registra los identificadores de ambos procesos, y la dirección y tamaño de la variable origen.
2. Inicio de lectura (IL): el SPM lee el primer byte de todos los que forman el valor de la variable origen.
3. FIn de lectura (FL): el SPM lee el último byte de todos los que forman el valor de la variable origen.

Instantes críticos en el lado del receptor En el lado del receptor, el SPM debe dar estos pasos: 1. Fin del registro de la solicitud de recepción (SR): después de iniciado receive, el SPM registra los identificadores de procesos y la dirección y tamaño de la variable de destino. 2. Fin del emparejamiento (EM): el SPM espera hasta que se haya registrado una solicitud de envío que case con la recepción anterior. Entonces se emparejan ambas. 3. Inicio de escritura (IE): el SPM escribe en la variable de destino el primer byte recibido. 4. Fin de escritura (FE): el SPM escribe en la variable de destino el último byte recibido.

Sincronización en el SPM para la transmisión La transmisión de mensajes supone sicronización:

- El emparejamiento solo puede completarse después de registrada la solucitud de envío, es decir, SE $<$ EM
- Antes de que se escriba el primer byte en el receptor, se debe haber comenzado ya la lectura en el emisor, por tanto IL $<$ IE.
- Antes de que se acaben de escribir los datos en el receptor, se deben haber acabado de leer en el emisor, es decir FL $<$ FE.
- Por transitividad, IL$<$FE.

Sin embargo no hay orden entre estas acciones: - No hay orden predefinido entre SE y SR. - No hay orden predefinido entre EM y IL (ni FL).

Se puede iniciar la lectura (IL) antes de que ocurra el emparejamiento (EM). Si esto se hace: - El SPM deberá almacenar temporalmente algunos o todos los bytes de la variable origen en alguna zona de memoria (en el lado del emisor). - Esa zona se llama almacén temporal de datos.

Mensajes en tránsito. Memoria Por la hipótesis de progreso finito, el intervalo de tiempo entre la solicitud de envío y el fin de la escritura tiene una duración no predecible. Estre SE y FE se dice que el mensaje está en tránsito. El SPM necesita usar memoria temporal para todos los mensajes en tránsito que esté gestionando en un momento dado. La cantidad de memoria necesaria dependerá de diversos detalles (tamaño y número de los mensajes en tránsito, velocidad de la transmisión de datos, política de envío de mensajes, etc). Dicha memoria puede estar ubicada en el nodo emisor y/o en el receptor y/0 en nodos intermedios, si los hay. En un momento dado, el SPM puede detectar que no tiene suficiente memoria para almacenamiento en el emisor hasta asegurarse de que hay memoria para enviar los datos.

Las operaciones podrían no ser seguras: el valor que el emisor pretendía enviar podría no ser el mismo que el receptor recibe:

- Operación de envío recepción segura: se puede garantizar a priori el valor de var_orig antes del envío (antes de SE) coincidirá con el valor de var_dest tras la recepción (después de FE)
- Operación de envío-recepción insegura se da en dos casos:
  - Envío inseguro: ocurre cuando es posible modificar el balor de var_orig entre SE y FL.
  - Recepción insegura: ocurre cuando se puede acceder a var_dest entre SR y FE, si se lee antes de recibirlo totalmente o se modifica después de haberse recibido parcialmente.
- Operaciones seguras:
  - Devuelven el control cuando se garantiza la seguridad: send no espera a la recepción, receive sí espera.
  - Existen dos mecanismos para el paso de mensajes seguro: envío y recepción síncronos y envío asíncrono seguro.
- Operaciones inseguras:
  - Devuelven el control inmediatamente tras hacerse la solicitud de envío o recepción, sin garantizar la seguridad.
  - EL programador debe asegurar que no se alteran las variables mientras el mensaje está en tránsito.
  - Existen sentencias adicionales para comprobar el estado de la operación.

Operaciones síncronas. Comportamiento s_send(variable_origen,ident_proceso_receptor)

Realiza el envío de los datos y espera bloqueado hasta que los datos hayan terminado de leerse en el emisor y se hayan iniciado y emparejado un reveive en el receptor. s_send no termina antes de que ocurran FL y EM.

receive(variable_destino, ident_proceso_emisor)

Espera bloqueado hasta que el emisor emita un mensaje con destino al proceso receptor (si no lo había hecho ya) y hasta que hayan terminado de escribirse los datos en la zona de memoria designada en la variable destino. receive no termina antes de que ocurra FE.

Las oreraciones síncronas exigen cita entre emisor y receptor: la operación s_send no devuelve el control hasta que el receive correspondiente sea alcanzado por el receptor. El intercambio de mensaje constituye un punto de sincronización entre emisor y receptor. El emisor podría hacer aserciones acerca del estado del receptor.

Las operaciones síncronas son fáciles de implementar pero poco flexibles. Solo son adecuadas cuando send y receive se inicial al mismo tiempo aproximadamente. Es necesario alternar llamadas en intercambios.

Envío asíncrono seguro send (variable, id_proceso_receptor)

Inicia el envío de los datos designados y espera bloqueado hasta que hayan terminado de copiarse todos los datos de variable a algún lugar seguro. Tras la copia de los datos designados, devuelve el control sin que tengan que haberse recibido los datos en el receptor. Por tanto, se devuelve el control después de FL. Se suele usar junto con la recepción síncrona (receive).

En definitica, el fin de send no depende de la actividad del receptor. Puede ocurrir antes, durante o después de la recepción.

Ventajas: - El uso de send lleva en general a menores tiempos de espera bloqueada que s_send, ya que no es necesario esperar el emparejamiento. - Usar send es generalmente más eficiente en tiempo y preferible cuando el emisor no tiene que esperar la recepción.

Inconvenientes: - send requiere memoria para almacenamiento temporal, la cual, en algunos casos, puede crecer mucho o indefinidamente. - El SPM puede tener que tretrasar el inicio de lectura (IL) en el lado del emisor, cuando ddtecta que no dispone de memoria suficiente para copiar los bytes y no se ha producido aún el emparejamiento con ningún receptor. - Aunque send sea asíncrono, si se utiliza con receive puede producir interbloqueo.

Operaciones inseguras Las operaciones seguras son menos eficientes, en tiempo y en memoria. La alternativa son las operaciones de inicio de envío o recepción: devuelven el control antes de que sea seguro modificar (envío) o leer datos (recepción).

Deben existir sentencias de chequeo de estado_ indican si los datos pueden alterarse o leerse sin comprometer la seguridad.

Una vez iniciada la operación, el usuario puede realizar cualquier cómputo que no dependa de la finalización de la operación y , cuando sea necesario, chequeará su estado.

Operaciones:

i_send (variable_orig, id_proc_receptor, var_resguardo)

Indica al SPM que comience una operación de envío al receptor. Se regista la solicitud de envío (SE) y acaba. NO espera a FL ni a ninguna acción del receptor. var_resguardo permite consultar después el estado del envío.

i_receive(var_dest, id_proc_emisor, var_resguardo)

Indica al SPM que se inicie una recepción de un mensaje del emisor. Se registra la solicitud de receción (SR) y acaba. No espera a FE ni a ninguna acción del emisor. var_resguardo permite consultar después el estado de la recepción.

Cuando un proces hace i_send o i_receive puede continuar trabajando hasta que llega el momento en que debe esperar a que termine la operación. Se disponen de estos dos procedimientos:

wait_send(var_resguardo)

Se invoca por el proceso emisor, y lo bloquea hasta que la operación de envío asociada a var_resguardo ha lelgado al instalte FL.

wait_recv(var_resguardo)

Se invoca por el proceso receptor, que queda bloequeado hasta que la operación de recepción asociada a var_resguardo haya llegado al instante FE.

Estas operaciones perminten a los procesos emisor y receptor hacer trabajo útil concurrentemente con la operación de envío y recepción. Mejora el tiempo de espera ociosa que se puede emplear en computación a cambio de una reestructuración del programa, más complejo de programar.

Limitaciones: - La duración del trabajo útil podría ser muy distinta que la de cada transmisión. - Se descarta la posibilidad de esperar más de un posible emisor.

Para solventar estos problemas se pueden usar dos funciones para comprobación de estado de transmisión de un mensaje. No suponen bloqueo:

test_send(var_resguardo)

Función lógica que se invoca por el emisor. Si el envío asociado a var_resguardo ha llegado al fin de la lectura (FL), devuelve true, si no devuelve false.

test_recv(var_resguardo)

Función lógica que se invoca por el receptor. Si el envío asociado a var_resguardo ha llegado al fin de la escritura (FE), devuelve true. Si no devuelve false.

Usando la opreación i_receive junto con las de test, se usa espera ocupada, de forma que_ - Se espera un mensaje cualquiera proveniente de varios emisores. - Tras recibir el primero de los mensajes se ejecuta una acción, independientemente de cual sea el emisor de ese mensaje. - Entre la recepción del primer mensaje y la acción el retraso es normalmente pequeño.

Sin embargo, con las primitivas vistas, no es posible cumplir estos requisitos usando espera bloqueada. Es inevitable seleccionar de antemano de que emisor queremos esperar recibir en primer lugar, y este emisor no coincide necesariamente con el emisor del primer mensaje que realmente podría recibirse.

1.4 Espera selectiva Espera selectiva es una operación que perminte espera bloqueada de múltiples emisores. Se usan las palabras clave select y when.

Para aplicarlautilizamos un proceso intermedio entre emisor y receptor, buffer.

Una guarda es ejecutable en un momento de la ejecuación de un porceso P cuando se dan las condiciones: - La condición de la guarda se evalúa en ese momento a true. - SI tiene sentencia de entrada, entonces el proceso origen nombrado ya ha iniciado en ese momento una sentencia send con destino al proceso P, que casa con el receive.

Una guarda será potencialmente ejecutable si se dan estas dos condiciones: - La condición de la guarda se evalúa a true. - Tiene una sentencia de entrada, sentencia que nombra a un proceso que no ha iniciado aún un send hacia P.

Una guarda será no ejecutable en el resto de casos, en los que forzosamente la condición de la guarda se evalúa a false.

Para ejecutar select, al inicio se selecciona una alternativa: - Si hay guardas ejecutables con sentencias de entrada se selecciona aquella cuyo send se inició antes. - Si hay guardas ejecutables, pero ninguna tiene una sentencia de entrada se selecciona aleatoriamente una cualquiera. - SI no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables se espera bloqueado a que alguno de los procesos nombrados en esas guardas inicie un send, en ese momento acaba la espera y se selecciona la guarda correspondiente a ese proceso. - Si no hay guardas ejecutables ni potencialmente ejecutables no se selecciona ninguna guarda.

Una vez se ha intentado seleccionar la guarda: - Si no se ha podido, no se hace nada y se finaliza la ejecución de select. - Si se ha podido, se dan estos dos pasos en la secuencia: 1. Si esa guarda tiene sentencia de entrada, se ejecuta el receive y se recibe el mensaje. 2. Se ejecuta la sentencia asociada a la alternativa y después finaliza la ejecución de select.

Hay que tener en cuenta que select conlleva potencialmente esperas, y por tanto se pueden producir esperas indefinidas (interbloqueo).

1.3.2 2. Paradigmas de interacción de procesos en programas distribuidos

2.1 Introducción Paradigma de interacción define un esquema de interacción entre procesos y una estructura de control que aparece en múltiples programas: - Unos pocos paradigmas de interacción se utilizan repetidamente para desarrollar muchos programas distribuidos. - Se usan principalmente en programación paralela, excepto el cliente servidor que es más general.

2.2 Maestro-esclavo En este aptrón de interacción intervienen dos entidades: un procedimiento maestro y múltiples procesos esclavos. El proceso maestro descompone el problema en subtareas y las distribuye entre los esclavos. Va recibiendo resultados parciales para producir el resultado final. Los procesos esclavos ejecutan un ciclo muy simple hasta que el maestro informa del final del

cómputo.

2.3 Iteración síncrona En múltiples problemas numéricos un cálculo se repite y cada vez que se obtiene se utiliza el resultado para el siguiente cálculo. El proceso se repite hasta obtener los resultados deseados. A menudo se pueden realizar los cálculos de cada iteración de manera concurrente. En un bucle diversos procesos comienzan juntos en el inicio de cada iteración. La siguiente iteración no puede comenzar hasta que todos los procesos hayan acabado la previa. Los procesos suelen intercambiar información en cada iteración.

2.4 Encauzamiento (pipelining) El problema se divide en una serie de tareas que se han de completar después de otra. Cada tarea se ejecuta por un proceso separado. Los procesos se organizan en un cauce dnde cada proceso se corresponde con una etapa del acuce y es responsable de una tarea particular. Cada etapa del cauce contribuirá al problema global y devuelve información que es necesaria para etapas posteriores del cauce. Patrón de comunicación muy simple ya que se establece un flujo de datos entre las tereas adyacentes en el cauce.

1.3.3 3. Mecanismos de alto nivel en sistemas distribuidos

3.1 Introducción. LLamada a procedimiento Los mecanismos vistos hasta ahora presentan un bajo nivel de abstracción. Veremos mecanismos de mayor nivel de abstracción: - Llamada a procedimiento remoto (RPC) - Invocación remota de métodos (RMI)

Ambos están basados en el método habitual por el cual un proceso llamador hace una llamada a procedimiento: 1. El llamador indica el nombre del procedimiento y los valores de los parámetros. 2. El proceso llamador ejecuta el código del procedimiento. 3. Cuando el procedimiento termina, el llamador obtiene los resultados y continúa ejecutando el código tras la llamada.

En el modelo de invocación remota o llamada a procedimiento remoto se dan los mismos pasos, pero es otro procedimiento (el proceso llamado) el que ejecuta el código del procedimiento. 1. El llamador indica el nombre del procedimiento y los valores de los parámetros. 2. El proceso llamador se queda bloqueado. EL proceso llamado ejecuta el código del procedimiento. 3. Cuando el procedimiento termina, el llamador obtiene los resultados y continúa ejecutando el código tras la llamada.

RPC representa estas características: - El flujo de comunicaciones es bidireccional. - Se permite que varios procesos invoquen un procedimiento gestionado por otro proceso.

3.2 El paradigma cliente-servidor Es el paradigma más frecuente en programación distribuida. Hay una relación asimétrica entre dos procesos: cliente y servidor.

- Proceso servidor: gestiona un recurso y ofrece un servicio a otros procesos (clientes) para permitir que puedan acceder al recurso. Puede estar ejecutándose durante un largo periodo de tiempo, pero no hace nada útil mientras espera peticiones de los clientes.
- Proceso cliente: necesita el servicio y envía un mensaje de petición al servidor solicitando algo asociado al servicio proporcionado por el servidor.

1.4 Tema 4. Introducción a los sistemas de tiempo real

1.4.1 1. Concepto de sistemas de tiempo real. Medidas de tiempo y modelo de tareas.

1.1 Definición, tipos y ejemplo Sistemas de tiempo real constituyen un tipo de sistema en el que la ejecución del sistema se debe producir dentro de unos plazos de tiempo predefinidos para que funcione con suficiente garantía. En un sistema además concurrente será necesario que todos los procesos sobre un procesador o sobre varios se ejecuten en los plazos de tiempo predefinidos.

Habitualmente suele asociarse la denominación de sistema de tiempo real a los siguientes tipos de sistemas: - Sistema en línea: siempre está disponible, pero no se garantiza una respuesta en un intercalo de tiempo acotado. - Sistema interactivo. Suele ofrecer una respuesta en un tiempo acotado, aunque no importa si ocasionalmente tarda más. - Sistema de respuesta rápida: el sistema ofrece una respuesta en un tiempo acotado y lo más corto posible.

1.2 Propiedades de los sistemas de tiempo real Al depender la corrección del sistema de las restricciones temporales, los sistemas de tiempo real tienen que cumplir una serie de propiedades: - Reactividad - Predecibilidad - Confiabilidad

Reactividad: el sistema tiene que interaccionar con el entorno y responder de la manera esperada a los estímulos externos de un intervalo de tiempo previamente definido.

Predecibilidad: tener un comportamiento predecible implica que la ejecución del sistema tiene que ser determinista, y por lo tanto, se debe garantizar su ejecución dentro del plazo de tiempo definido. - Las respuestas han de producirse dentro de las restricciones temporales impuestas por el entorno, que suelen ser diferentes para cada proceso del sistema. - Es necesario conocer el comportamiento temporal de los componentes software y hardware utilizados, así como el lenguaje de programación. - Si no se puede tener un conocimiento temporal exacto, hay que definir marcos de tiempo acotados.

Confiabilidad: la confiabilidad mide el grado de confianza que se tiene en el sistema. Depende de: - Disponibilidad: capacidad de proporcionar servicios siempre que se solicita. - Robustez o tolerancia a fallos: capacidad de operar en situaciones excepcionales sin poseer un comportamiento catastrófico. - Fiabilidad: capacidad de ofrecer siempre los mismos resultados bajo las mismas condiciones.

- Seguridad: capacidad de protegerse ante ataques o fallos accidentales o deliberados, y no la no vulverabilidad de los datos. Cuando esta propiedad es crítica el sistema se denomina sistema de tiempo real cítico.

1.3 Modelo de Tareas Una tarea es un conjunto de acciones que describen el comportamiento del sistema o parte de él en base a la ejecución secuencial de un trozo de código. Equivalente a proceso o hebra. - La tarea satisface una necesidad funcional concreta. - La tarea tiene definida unas restricciones temporales a partir de los atributos temporales. - Una tarea activada es una tarea en ejecución o pendiente de ejecutar. - Decimos que una tarea se activa cuando se hace necesario ejecutarla una vez. - El instante de activación de una tarea es el instante de tiempo a partir del cual debe ejecutarse (cuando pasa de desactivada a activada).

Los recursos son elementos disponibles para la ejecución de las tareas. Se distinguen dos tipos de recursos: - Recursos activos: procesador, red... - Recuross pasivos: datos, ememoria, dispositivos de E/S...

Asumimos que cada CPU disponible se dedica a ejecutar una o varias tareas, de acuerdo al esquema de planificación en uso. Si una CPU ejecuta más de una tarea, el tiempo de la CPU debe repartirse entre varias tareas.

Los requisitos de un sistema de tiempo real obligan a asociar un conjunto de atributos temporales a cada tarea de un sistema. Estos atributos son restricciones acerca de cuando se ejecuta activa cada tarea y cuanto puede tardar en completarse desde que se activa.

La planificación de tareas es una labor de diseño que determina como se le asignan a lo largo del tiempo a cada tarea los recursos activos de un sistema, de forma que se garantice el cumplimiento de las restricciones dadas por los atributos temporales de la tarea.

Atributos temporales de una tarea: - Tiempo de cómputo o de ejecución (C): tiempo necesario para la ejecución de la tarea. - Tiempo de respuesta (R): tiempo que ha necesitado el proceso para completarse totalmente a partir del instante de activación. - Plazo de respuesta máxima (D): define el máximo de tiempo de respuesta posible. - Periodo (T): intervalo de tiempo entre dos actividades sucesivas en el caso de una tarea periódica.

Tipos de tareas según la recurrencia de sus actividades:

- Aperiódicas: se activan en instantes arbitrarios (no tiene T).
- Periódicas: repetitivas, T es el tiempo exacto entre activaciones.
- Esporádicas: reptitiva, T es intervalo mínimo entre activaciones.

Diseño de la planificación de tareas El problema de la planificación supone: - Determinar los procesadores disponibles a los que se puede asociar las tareas. - Determinar las relaciones de dependencia de las tareas: - Relaciones de precedencia que hay entre las distintas tareas. - Determinar los recursos comunes a

los que accenden las distintas tareas. - Determinar el orden de ejecución de las tareas para garantizar las restricciones especificadas.

Para determinar la planificabilidad de un conjunto de tareas se requiere un esquema de planificación que cubra los dos siguientes aspectos: - Un algoritmo de plafinicación, que define un criterio (política de planificación) que determina el orden de acceso de las tareas a los distintos procesadores. - Un método de análisis que permite predecir el comportamiento temporal del sistema, y determina si la planificabilidad es factible bajo las condiciones o restricciones especificadas. - Se pueden comprobar si los requisitos temporales están garantizados en todos los casos posibles. - En general es estudia el peor comportamiento posible, es deecir, con el WCET (Worst Case Execution Time).

Cálculo del WCET Suponemos que siempre se conoce el valor WCET (Cw) que es el máximo valor de C para cualquier ejecución posible de dicha tarea. Hay dos formas de obtener Cw: - Medida directa del tiempo de ejecución (en el peor caso) en la plataforma de ejecución. Se realizan múltiples experimentos, y se hace una estadística. - Análisis del código. Se basa en calcular el peor tiempo. Se descompone el código en un grafo de bloques secuenciales. Se calcula el tiempo de ejecución de cada bloque. Se busca el camino más largo.

Mientras que no se diga lo contrario, asumimos que siempre se tarda lo mismo (C) en ejecutar una tarea determinada. Por lo tanto se cumple $C = Cw$.

Restricciones temporales de una tarea Para determinar la planificación del sistema necesitamos conocer las restricciones temporales de cada tarea del sistema. Las restricciones temporales para un conjunto de n tareas periódicas se especifican dando una tabla con los valores Ti, Ci y Di para cada una de ellas. La i-ésima tarea ocupa una fracción Ci/Ti del tiempo total de una CPU. El factor de utilización U es la suma de esas fracciones para todas las tareas. En un hardware con p procesadores disponibles para ejecutar las tareas, si $U > p$ entonces el sistema no es planificable: incluso dedicando a ejecutar tareas el 100% del tiempo de cada uno de los p procesadores, alguna tarea no podrá acabar su período.

1.4.2 2. Esquemas de planificación

Tipos de esquemas de planificación Para un sistema monoprocesador son los siguientes: - Planificación estática off-line sin prioridades (ejecutivo cíclico) - Planificación basada en prioridades de tareas - Estática: prioridades preasignadas, no cambian. - RMS (Rate Monotonic Scheduling): prioridad a la tarea con menor período T. - DMS (Deadline Monotonic Scheduling): prioridad a la tarea con menor deadline D. - Dinámicas: prioridades cambiantes durante la ejecución - EDF (Earliest Deadline First): prioridad a la tarea con el deadline más próximo. - LLF (Least Laxity First): prioridad a tarea de menor holgura (tiempo hasta deadline menos tiempo de ejecución restante)

2.1 Planificación cíclica La planificación se basa en diseñar un programa (ejecutivo cíclico) que implementa un plan de ejecución (plan principal) que garantice los requerimientos temporales. El programa es un bucle infinito tal que cada iteración tiene una duración prefijada, siempre igual. EL bucle se denomina ciclo principal. En cada iteración del bucle principal se ejecuta otro bucle acotado con k iteraciones (k es constante). Cada iteración dura siempre lo mismo y en ella se ejecutan completamente una o varias tareas. A este bucle interno (acotado) se le denomina ciclo secundario. El extrelazado de las tareas en cada iteración del ciclo principal es siempre el mismo. Una iteración número i del ciclo secundario puede tener un entrelazado distinto a otra iteración número j (1<= i, j <=k).

Las duraciones de ambos ciclos son valores enteros (se supone que el tiempo se mide en múltiplos enteros de alguna unidad de tiempo). La duración del ciclo principal se denomina hiperperiodo y se escribe como T_M. EL hiperperiodo es el mínimo común múltiplo de los periodos de todas las tareas. Por tanto, los instantes de inicio de cada iteración del ciclo principal coinciden con los instantes en los cuales todas las tareas se activan de nuevo a la vez. La duración del ciclo secundario se denomina T_s. Se debe cumplir que T_M = kT_s.

Para seleccionar un valor apropiado para T_s se deben tener en cuenta estas restricciones y sugerencias: - Restricciones: necesariamente se cumplen estas dos - T_s es necesariamente divisor de T_M - El valor de T_s tiene que ser mayor o igual que el tiempo de cómputo (Ci) de cualquier tarea. - Sugerencia: es aconsejable intentar en principio que el ciclo secundario sea menor o igual que el mínimo deadline.

Propiedades de la plafinifación cíclica:

- No hay concurrencia en la ejecución. Cada ciclo secundario es una secuencia de llamadas a procedimientos. No se necesita un núcleo de ejecución multitarea.
- Los procedimientos pueden compartir datos. No se necesitan mecanismos de exclusión mutua como los semáforos o monitores.
- No hace falta analizar el comportamiento temporal. El sistema es correcto por construcción.

Problemas:

- Dificultad para incorporar tareas con periodos largos
- Las tareas esporádicas son difíciles de tratar. Se puede utilizar un servidor de consulta.
- El plan cíclico del proyecto es difícil de construir. Si los periodos son de diferentes órdenes de magnitud el número de ciclos secundarios se hace muy grande. Puede ser necesario partir de una tarea en varios procedimientos. Es el caso más general de sistemas en tiempo real críticos.
- Es poco flexible y difícil de mantener. Cada vez que se cambia una tarea hay que rehacer la planificación.

1.4.3 2. Planificación con prioridades

La planificación con prioridades permite solventar los problemas descritos. Cada tarea tiene asociado un valor entero positivo, llamado prioridad de la tarea. Es un atributo entero no negativo de las tareas, que depende de sus atributos temporales y/o el enlazamiento entre ellas. Por convención se asigna números enteros mayores a tareas más urgentes. Las prioridades pueden ser constantes fijadas de antemano (estáticas) o bien pueden cambiar con el tiempo (dinámicas), en este caso se deben calcular cada vez que hay que consultarlas.

En este tipo de planificaciones debe existir una componente software (scheduler) capaz de: - Asignar el procesador a una tarea activa o ejecutable (despachar la tarea). - Suspender una tarea en ejecución cuando es necesario. - Una tarea puede estar en viarios estados (suspendida, ejecutable, ejecutándose). - Las tareas ejecutables se despachan para su ejecución en orden de prioridad.

El planificador actúa en cada instante de tiempo en el que ocurren alguno de estos eventos: - Una o más tareas se activan (pasan a ejecutable). - La tarea en ejecución termina (pasa al estado de suspendida).

A continuación, selecciona cualquier tarea A con una prioridad actual P_A máxima entre todas las ejecutables. Después: - Si la CPU está libre, A pasa a ejecutándose. - Si la CPU está ejecutando una tarea B con prioridad actual P_B: - Si P_A > P_B la tarea A pasa a estado ejecutándose y B pasa a estado suspendido. - Si P_A <= P_B no hay cambios.

Al inicio, todas las tareas se activan a la vez.

Planificación RMS Rate Monotonic Scheduling es un método de planificación estático on-line con asignación mayor priotidad a las tareas más frecuentes (con menor período). - A cada tarea i se le asigna una prioridad P_i basada en su periodo. Cuanto menor sea el período mayor su prioridad. - Esta asignación de prioridades es óptima en el caso de que todas las tareas sean periódicas, y el plazo de respuesta máxima D coincida con el periodo.

Test de planificabilidad Los test de planificabilidad permiten determinar si el conjunto de tareas del sistema es planificable según un algoritmo de planificación antes de su ejecución. Existen diversos tipos de test aplicables según el algoritmo de planificación: - Test de planificación suficientes: en caso de exito en la aplciación del test el sistema es planificable. En caso contrario no tentmos información, podría ser planificable o no. - Test de planificación exactos: en caso de exito la aplicación es planificable. En caso contrario el sistema no es planificable.

El test de Liu & Layland es un test suficiente, determina la planificabilidad de un sistema de n tareas periódicas independientes con prioridades RMS. Se usan dos valores realies, el factor de utilización (U) y el factur de utilización máximo (U_0(n)) para n tareas.

Un sistema pasa el test si el factor de utilización es menor o igual que el

máximo posible. En ese caso el sistema es planificable. En caso contrario no se puede afirmar nada.

Si una planificación no pasa el test hay que hacer el cronograma y verificar que: - para cada tarea i se cumple el plazo de respuesta R < D - esto se debe verificar para un hiperperiodo.

Planificación EDF Es un esquema de planificación con prioridades dinámicas. Se denomina EDF, o bien primero el más urgente. La asignación de prioridad se establece una prioridad más alta a la que se encuentre más próxima a su deadline. Características: - En caso de igualdad se hace una elección no determinista de la siguiente tarea a ejecutar. - Es un algoritmo de planificación dinámica, dado que la prioridad de cada tarea cambia durante la evolción del sistema. - Es más óptimo porque no es necesario que las tareas sean periódicas. - Es menos pesimista que RMS.

El test de planificación Liu & Layland se puede aplicar también a EDF. En este caso un sistema pasa el test si el factor de utilización es igual o menor que la unidad. Esto implica que la planificación EDF puede aplicarse a más sistemas que la planificación RMS. El test en este caso es exacto.