seguridad y vivacidad/viveza

Un programa concurrente puede fallar por varias razones, las cuales se pueden clasificar entre dos grupos de propiedades:

seguridad: Esa propiedad indica que no está pasando nada malo en el programa, es decir, el programa no ejecuta instrucciones que no deba hacer (“safety property”).

vivacidad: Esa propiedad indica que está pasando continuamente algo bueno durante la ejecución, es decir, el programa consigue algún progreso en sus tareas o en algún momento en el futuro se cumple una cierta condición (“liveness property”).

propiedades de seguridad

Las propiedades de seguridad suelen ser algunas de las invariantes del programa que se tienen que introducir en las comprobaciones del funcionamiento correcto.

Corrección: El algoritmo usado es correcto.

Exclusión mutua: El acceso con exclusión mutua a regiones críticas está garantizado

Sincronización: Los procesos cumplen con las condiciones de sincronización impuestos por el algoritmo

Interbloqueo: No se produce ninguna situación en la cual todos los procesos participantes quedan atrapados en una espera a una condición que nunca se cumpla.

propiedades de vivacidad

inanición

- Un proceso puede “morirse” por inanición (*starvation*), es decir, un proceso o varios procesos siguen con su trabajo pero otros nunca avanzan por ser excluidos de la competición por los recursos (por ejemplo en Java el uso de `suspend()` y `resume()` no está recomendado por esa razón).
- Existen problemas donde la inanición no es un problema real o es muy improbable que ocurra, es decir, se puede aflojar las condiciones a los protocolos de entrada y salida.

propiedades de vivacidad

Bloqueo activo: Puede ocurrir el caso que varios procesos están continuamente competiendo por un recurso de forma activa, pero ningúno de ellos lo consigue (“livelock”).

Cancelación: Un proceso puede ser terminado desde fuera sin motivo correcto, dicho hecho puede resultar en un bloqueo porque no se ha considerado la necesidad que el proceso debe realizar tareas necesarias para liberar recursos (por ejemplo, en Java el uso del `stop()` no está recomendado por esa razón).

Espera activa: Un proceso está comprobando continuamente una condición malgastando de esta manera tiempo de ejecución del procesador.

justicia entre procesos

Cuando los procesos compiten por el acceso a recursos compartidos se pueden definir varios conceptos de justicia, por ejemplo:

justicia débil: si un proceso pide acceso continuamente, le será dado en algún momento,

justicia estricta: si un proceso pide acceso infinitamente veces, le será dado en algún momento,

espera limitada: si un proceso pide acceso una vez, le será dado antes de que otro proceso lo obtenga más de una vez,

espera ordenada en tiempo: si un proceso pide acceso, le será dado antes de todos los procesos que lo hayan pedido más tarde.

comentarios

- Los dos primeros conceptos son conceptos teóricos porque dependen de términos *infinitamente* o *en algún momento*, sin embargo, pueden ser útiles en comprobaciones formales.
- En un sistema distribuido la ordenación en tiempo no es tan fácil de realizar dado que la noción de tiempo no está tan clara.
- Normalmente se quiere que todos los procesos manifesten algún progreso en su trabajo (pero en algunos casos inanición controlada puede ser tolerada).

espera activa de procesos

- El algoritmo de Dekker y sus parecidos provocan una espera activa de los procesos cuando quieren acceder a un recurso compartido. Mientras están esperando a entrar en su región crítica no hacen nada más que comprobar el estado de alguna variable.
- Normalmente no es aceptable que los procesos permanezcan en estos bucles de espera activa porque se está gastando potencia del procesador inútilmente.
- Un método mejor consiste en suspender el trabajo del proceso y reanudar el trabajo cuando la condición necesaria se haya cumplido. Naturalmente dichos técnicas de control son más complejas en su implementación que la simple espera activa.

evitar espera infinita o inanición de procesos

- Se implementa, por ejemplo, el acceso a recursos compartidos siguiendo un orden FIFO, es decir, los procesos tienen acceso en el mismo orden en que han pedido vez.
- Se asignan prioridades a los procesos de tal manera que cuanto más tiempo un proceso tiene que esperar más alto se pone su prioridad con el fin que en algún momento su prioridad sea la más alta.
- ¡Ojo! ¿Qué se hace si todos tienen la prioridad más alta?
- Existen más técnicas... ¿Cuáles?

comunicación

Programas concurrentes o/y distribuidos necesitan algún tipo de comunicación entre los procesos.
Hay dos razones principales:

1. Los procesos compiten para obtener acceso a recursos compartidos.
2. Los procesos quieren intercambiar datos.

metódos de comunicación

Para cualquier tipo de comunicación hace falta un método de sincronización entre los procesos que quieren comunicarse entre ellos.
Al nivel del programador existen tres variantes como realizar las interacciones entre procesos:

1. Usar memoria compartida (*shared memory*).
2. Mandar mensajes (*message passing*).
3. Lanzar procedimientos remotos (*remote procedure call* RPC).

síncrono y asíncrono

- La comunicación no tiene que ser síncrona en todos los casos.
- Existe también la forma asíncrona donde un proceso deja su mensaje en una estructura de datos compartida por los procesos.
- El proceso que ha mandado los datos puede seguir con otras tareas.
- El proceso que debe leer los datos, lo hace en su momento.

canal de comunicación

- Una comunicación entre procesos sobre algún canal físico puede ser *no fiable* en los sistemas.
- Se puede usar el canal
  - para mandar paquetes individuales del mensaje (por ejemplo protocolo UDP del IP)
  - para realizar flujos de datos (por ejemplo protocolo TCP de IP)
- Muchas veces se realiza los flujos con una comunicación con paquetes añadiendo capas de control (pila de control).

posibles fallos en canales de paquetes

Para los canales de paquetes, existen varias posibilidades de fallos:

1. se pierden mensajes entre transmisor y receptor
2. se cambia el orden de los mensajes (es decir, se reciben en otro orden que enviados)
3. se modifican mensajes (es decir, se altera su contenido)
4. se añaden mensajes que nunca fueron mandados

técnicas para superar los problemas

1. protocolo de recepción
   (¿Cuándo se sabe que ha llegado el último mensaje?)
2. enumeración de los mensajes
3. uso de código de correción de errores (CRC)
4. protocolo de autenticación

comunicación segura sin mensajes de asentimiento

- Existen protocolos de transmisión de paquetes que no necesitan un canal de retorno pero que garantizan la distribución de los mensajes bajo leves condiciones al canal (*digital fountain codes*).
- Ejemplos: Reed/Solomon códigos (clásicos), Tornado códigos (finales de los 90), Raptor códigos (*rap*id *tom*ado, en los últimos años))
- Raptor código: ejemplo, se manda unos 1.002 MByte, y de cualquier 1 MByte recibido se puede recuperar el mensaje original (con tiempo de cálculo lineal en la longitud del mensaje)
- base para varios nuevos estandares de transmisión de datos en la red

problemática específica

- terminación distribuida
- gestión de memoria distribuida
- estado distribuido
- propiedades distribuidos
- tiempo distribuido
- comunicación