Software crítico para sistemas espaciales

2024/25

Master Universitario en Enxeñaría Aeroespacial

Arno Formella
Miguel Ramón González Castro
Manuel Pérez Cota

Departamento de Informática
Universidade de Vigo

24/25

concurrencia

- hablamos de programación concurrente cuando varias entidades actuan lógicamente en paralelo
- dicho funcionamiento en paralelo puede llevarse a cabo en hardware paralelo donde las secuencias de instrucciones por ejecutar concurrentemente se planifican sobre cierto número de unidades de procesamiento
- ejemplos son: programación multi-hilo, programación multi-proceso, programación distribuida

concurrencia: problema de escritura concurrente

- las entidades actuando concurrentemente suelen intercambiar información mediante objectos compartidos
  - escriben y leen en una memoria compartida
  - actuan con un mensaje al remoto cambiando el estado de un objeto remoto
- la lectura de datos compartidos no suele provocar problemas
- la escritura sí puede traer problemas de consistencia:
  - o bien ningún participante tiene éxito escribiendo
  - o bien cualquier de los participantes tiene éxito escribiendo
  - o bien el resultado es aleatorio

concurrencia: concepto de evento

- podemos describir las acciones que realiza una entidad sobre un dato/objeto/recurso compartido como evento
- si ninguna acción de otro participante influye en la semántica de tal evento, llamamos el evento atómico
- un participante en un programa concurrente realiza (de vez en cuando según algoritmo) un tal evento sobre los datos/objetos/recursos compartidos
- además muchos algoritmos concurrentes requieren que una secuencia de eventos se ejecute sin posible influencia de los demás (transacción o secuencia en exclusión mutua)
- ejemplo de cuenta bancaria:
  leer cuenta, actualizar saldo, escribir saldo

concurrencia: secuencias de eventos

- a parte que algún participante quiere ejecutar secuencias de eventos con exclusión mutua
- existen algoritmos donde se require que eventos realizados por participantes diferentes se ejecutan en un orden lógico específico
- ejemplo: un hilo abre un fichero, otros hilos escriben en tal fichero, finalmente otro hilo cierre tal fichero

concurrencia: exclusión mutua

- la exclusión mutua se puede realizar con cerrojos que solo un participante adquiere y los demás están forzados a esperar hasta que lo suelte de nuevo
- en principio se puede implementar cerrojos con simples lecturas y escrituras atómicas a variables compartidas
- existen, en muchos sistemas, instrucciones más potentes para realizar cerrojos (diferentes tipos de *read-modify-write-operations*)
- sin entrar en detalles: existen algoritmos y mecanismos que realizan concurrencia sin cerrojos, llamados *lock-free* y *wait-free*

concurrencia: problemas de exclusión mutua

- si varios participantes actuan con exclusión mutua sobre varios recursos pueden aparecer dos tipos de problemas
  - no se consigue acceso a los recursos necesarios
  - no se ejecutan las secuencias en orden requerido
- error de bloqueo total o parcial (*deadlock*)
  dos o más participantes están mutuamente esperando a algo que nunca ocurre
- errores de sincronización (*race condicion* o condición de carrera)
  dos o más eventos no están ocurriendo en el orden temporal o lógico requirido

linearizabilidad (*linearizability*)

- linearizabilidad (de operaciones) es una garantía que cierto tipo de operaciones/eventos sobre un único objeto ocurren realmente en el orden temporal como especificado
- eso significa que se puede asumir del punto de vista lógico que tales operaciones son
  - atómicos en su ejecución
  - todo efecto está visible para operaciones posteriores
  - se puede secuenciar de forma unica las operaciones

serializabilidad (*serializability*)

- serializabilidad (de transacciones) es una garantía que ciertas transacciones (secuencias de operaciones/eventos) no interfieren entre si mismas, sino, que las transacciones se ejecutan completamente e independientemente aunque en un orden no predeterminado (si hay concurrencia física en el sistema)
- si las transacciones son asociativas, el orden de ejecución efectivamente no importa
- serializabilidad se consigue, por ejemplo, con
  - ejecución de forma serial garantizado (mono-proceso)
  - uso de cerrojos para garantizar exclusión mutua
  - detección de inconsistencia y re-ejecución de la transacción

modelos de consistencia de concurrencia

- linearizabilidad $\neq$ serializabilidad
- conseguir las dos propiedades facilita hasta cierto punto la implementación de aplicaciones/algoritmos concurrentes
- existen otros modelos de consistencia de concurrencia que son menos fuertes (pero quizá no adecuado para sistemas críticos)
- ejemplos en Java no-atómicos (increment, read/write de valores de 64-bit, pero hay bibliotecas/paquetes)

estrategias de comprobación especiales

- comprueba para cada variable y dato si es un valor compartido o privado
- añade la noción de tiempo a la necesidad de tener un dato de forma privado, eso nos aclara si hace falta un mecanismo de sincronización
- *single assignment* posiblemente aumenta la visibilidad de concurrencia

pruebas específicas para programas concurrentes

- un programa concurrente debe funcionar con cualquier demora introducida en cualquier hilo, incluye eso en las pruebas, p.ej, introducir demoras aleatorias en puntos aleatorios (excluidos programas de tiempo real)
- un programa concurrente debe funcionar con cualquier planificador (suficientemente justo), incluye eso en las pruebas, p.ej. ejecutar el código en diferentes entornos (hardware, sistema operativo, maquina virtual)

condiciones de carrera

- carrera sobre datos
  acceso concurrente a variables compartidas de los cuales por lo menos uno es una escritura
- error de atomicidad
  una secuencia de eventos no se ejecuta sin intervención de otro, es decir, otro eventos se intercalan
- error en secuencia
  una serie de eventos no se ejecutan en el orden requirido según especificaciones

posibles problemas subyacentes con pruebas

- no todos los hardware se comportan iguales, es decir, puede ser que ciertas intercalaciones de operaciones concurrentes posibles no son observables en todos los sitemas
- otras capas del sistema (interrupciones, sistema operativo, carga de aplicaciones concurrentes) pueden influir en las intercalaciones de operaciones observables
- programas concurrentes no son fácilmente divisibles en componentes, ya que requisitos de sincronización crucen el concepto de componente aislado, es decir, incluso componentes correctos (o probadas con pruebas unitarias) no necesariamente componen un sistema concurrente correcto global

como tratar posibles bloqueos

- prevenir
  por ejemplo, se adquiere todos los cerrojos entre todos los procesos siempre siempre según un orden preestablecido, así no se pueden formar ciclos con bloqueo resultante
- evitar
  se encarga a una entidad central de asignar los recursos (por ejemplo, cerrojos) de tal manera que no se produczan bloqueos
- detectar y actuar
  una entidad central (con las estructuras de datos y protocolos adecuados) observan si se ha producido un bloqueo, una vez detectado, se puede actuar para salvar la situación

algoritmos sin cerrojos

- basándose en ciertas instrucciones de *leer-y-modificar-de-forma-atómica* a nivel hardware se pueden implementar mecanismos de sincronización sin uso de cerrojos
  - algoritmos sin uso de cerrojos (*lock-free*)
  - algoritmos con avanza garantizado (*wait-free*)
    es decir, se garantiza que cualquier proceso tiene un avance en un número finito de pasos
- hay que tener sumo cuidado con compiladores modernos que reordenan instrucciones, y procesadores modernos que replanifican las ejecuciones de instrucciones con una semántica más permisiva
- suelen existir mecanismos de prevención de tales reordenaciones (*memoria barriers*)

eficiencia de algoritmos sin cerrojos

- la eficiencia de algoritmos sin uso de cerrojos suele ser por lo menos lineal en el número de participantes
- que en ciertas circunstancias eleva el tiempo de cálculo
- existen implementaciones de estructuras de datos básicos (p.ej. FIFO) que se pueden operar en modo *wait-free* y su rendimiento es comparable con métodos que usen cerrojos (depende del caso en concreto)

razones frecuentes del fracaso de proyectos software

las razones más frecuentes de mala calidad de software son:

- no está claro lo que hay que hacer
  (software requirement specification incomplete)
- equipo de desarrollo no adecuado
  (zapatero para zapatos, informáticos para la informatica)
- control de calidad deficiente
  (software product assurence program deficiente)
- falta de conocimientos técnicos
  (se necesita expertos para trabajos de calidad)
- subestimación de condiciones iniciales
- insuficiente participación/involucración del cliente final

costes una vez haber encontrado un fallo

se estima que los costes para arreglar un fallo encontrado en el software aumentan cuando más tarde en el proceso pasa:

- 1x si es en fase de requisitos/especificaciones
- 5x si es en fase de codificación y pruebas de unidad
- 10x si es en fase de integración y pruebas funcionales
- 15x si es en fase de pruebas betas (ya a nivel clientes seleccionados)
- 30x si es en fase de ya desplegado el software
- estimación 2020: 2 billiones de dolares en EE.UU.

¡para software crítico estas estimaciones son demasiado bajas!

factores que influyen en el proceso del desarrollo de software

- presión temporal (plazos a cumplir)
  (tuvimos suerte en XaTcobeo)
- presión económica (presupuestos disponibles)
- requisitos de compatibilidad (reusabilidad)
- integración de bibliotecas externas (drivers)
  (ha pasado en XaTcobeo)
- uso de código antiguo
  (pasado en Ariane5)
- disponibilidad de expertos y expertos en informática

modularización

para generar software fiable y robusto se necesita *craftsmanship* (artesania), es decir:

- formación y experiencia
- junto con dedicación y trabajo

mantenimiento de software

- software no es un coche (Lamport)
- un coche hay que mantener durante su uso
- software no hay que mantener (como tal)
- un coche tiene que ser funcional
- software tiene que ser correcto

mantenimiento de software

- se refiere a adaptar a nuevos entornos
- por ejemplos cambios en plataformas, sistemas operativos, compiladores, lenguaje de programación, interfaces, usuarios
- se refiere a una posible refactorización para simplificar o mejorar
- por ejemplo para reusabilidad de componentes, mejora en rendimiento

métricas

- lines of code per function
- lines of code per module/file
- number of methods per class
- number of atributes per class
- number of arguments to functions
- number of classes per library
- number of commits per development fase
- number of test cases considered

métricas

- number of global variables
- number of type casts
- number of declaration of same item
- nesting level of loops
- nesting level of if
- nesting level of blocks in general
- logical expression complexity in conditions
- call graph properties
- exception graph properties
- dependency graph properties (classes, files, libraries, etc.)

mantener un ambiente positivo

- fallos ocurren
- felicitar a aquellos que han encontrado un error en un software, incluso que fue en tu código
- mantener cierto nivel de humor y ambiente alegre en el equipo
- mantener las anéctotas para las reuniones informales especialmente con gente nueva en el equipo
- incorporar en la formación no solo la parte lo que hay que hacer, sino, lo que no hay que hacer, de *malos* ejemplos se aprende igual de bien...

como último

recuerda:

¡lo que no está escrito, basicamente no existe!

SCRISE, *the writings*.