Appunti sistemi embedded
Scheduling Real-Time Systems

2025

Contents

1 Introduzione

I sistemi real time sono sistemi di computazione dove la reazione a eventi dell'ambiente deve avvenire entro precisi requisiti temporali. Dunque la correttezza della computazione non dipende unicamente dal risultato prodotto, ma anche dal tempo in cui il risultato è prodotto. Una reazione ritardata potrebbe essere inutile o addirittura dannosa. Molto spesso questi sistemi vengono implementati attraverso la programmazione in assembly o con driver a basso livello per la gestione delle periferiche, manipolazione delle task e la priorità degli interrupt.
Questo approccio presenta degli svantaggi

- Programmazione tediosa
- Codice poco comprensibile
- Difficile manutenibilità
- Difficile verifica dei vincoli temporali

La conseguenza maggior di questo approccio è che il software costruito tramite tecniche empiriche può essere impredicibile.
Se tutti i vincoli temporali critici non possono essere verificati a priori e il sistema operativo non include meccanismi specifici per maneggiare i task real-time il sistema potrebbe apparentemente funzionare correttamente per un certo periodo di tempo, ma potrebbe fallire in situazioni particolari. Il testing, pur essendo importante, non può essere una soluzione per ottenere la predicibilità in un sistema real-time. Questo vale perché il flusso di controllo dipende anche da eventi esterni dell'ambiente, che non possono essere interamente inclusi nel testing.

2 Real time

La correttezza del sistema non dipende solamente dal risultato logico prodotto dalla computazione ma anche dal tempo in cui il risultato è prodotto. Il termine reale indica che la reazione del sistema a eventi esterni deve avvenire **durante** la loro evoluzione. Di conseguenza il tempo interno al sistema deve essere misurato usando la stessa scala utilizzata per misurare il tempo dell'ambiente controllato. Al livello di processo la principale differenza tra un task real-time e non è la presenza di una **deadline**, ovvero il tempo massimo in cui una task deve essere completata. In un sistema critico una task che non rispetta la deadline non solo è in ritardo, è anche un errore! In base alle conseguenze a cui porta il non rispettare la deadline le task possono essere suddivise in:

- **Hard** se il non rispetto della deadline porta a conseguenze catastrofiche, ad esempio il fallimento di un sistema di controllo di un aereo.
- **Firm** se il non rispetto della deadline porta a un output inutile al sistema ma non produce conseguenze catastrofiche.
- **Soft** se il non rispetto della deadline porta a una degradazione della qualità del servizio ma non produce conseguenze catastrofiche.

Un sistema in grado di gestire task hard real-time è detto **hard real-time system**. Alcuni di questi sistemi sono tendenzialmente sistemi **safety critical**.

2.1 Limiti dei sistemi real-time correnti

Molti dei sistemi attualmente in uso sono soluzioni basate su kernel, che sono versioni modificate di sistemi operativi a time sharing. Una conseguenza di questo è che condividono le stesse feature di base dei sistemi a time sharing, che non sono adatte a sistemi real-time. Le caratteristiche principali sono:

- **Multitasking**
  Supporto per la programmazione concorrente offerta attraverso un insieme di chiamate a sistema per la gestione dei processi. Molte di queste primitive non prendono in considerazione il tempo e, peggio ancora, introducono ritardi senza limiti ai tempi di esecuzione, il che potrebbe portare le task hard a non rispettare le deadline in maniera non predicibile.

- **Scheduling basato su priorità**
  Meccanismo flessibile di scheduling, dato che permette di implementare diverse strategie in base alla maniera in cui assegno le priori Nonostante ciò, mappare i constraint a un insieme di priorità è difficile e non sempre possibile, specialmente in un sistema dinamico. Il problema principale è che i kernel permettono un **numero limitato di livelli di priorità**, mentre le deadline possono variare molto di più. In sistemi dinamici l'arrvo di una nuova task può comportare un remappaggio dell'intero insieme di priorità.

- **Capacità di rispondere velocemente agli interrupt**
  Solitamente questa feature viene ottenuta tramite la possibilità di impostare una priorità per gli interrupt, più alta rispetto alla priorità dei processi e riducendo al minimo le porzioni di codice eseguite con le interruzioni disabilitate.
  Nota che questo approccio aumenta la reattività del sistema agli eventi esterni, ma introduce dei ritardi non limitati all'esecuzione dei processi. Infatti, un processo di un'applicazione sarà sempre interrotto da un interrupt di un driver, anche se è più importante che venga servita la periferica.

- **Meccanismo di base per la comunicazione tra processi e sincronizzazione**
  Tipicamente semafori binari per la sincronizzazione e la mutua esclusione su risorse condivise. Però, se nessun protocollo d'accesso è utilizzato per entrare in sezioni critiche i semafori possono generare una serie di problemi, ad esempio la **priority inversion** o deadlokc.

- **Small kernel e context switch veloce**
  Questa feature riduce l'overhead del sistema, dunque migliora il response time medio dell'insieme di task. Ciò nonostante non è sufficiente per garantire il rispetto delle deadline. D'altro canto un kernel piccolo implica limitate funzionalità, il che ha effetto sulla predicibilità del sistema.

- **Supporto a clock real-time come reference di tempo interna**
  Feature essenziale di ogni kernel real-time che gestisce task hard real-time che interagiscono con l'ambiente esterno. Nella maggior parte dei kernel commerciali questo è l'unico meccanismo di temporizzazione disponibile. In molti casi non vi sono primitive per controllare esplicitamente i constraint temporali delle task, e non c'è un supporto per l'attivazione automatica delle task periodiche.

La mancanza di una qualsiasi garanzia preclude l'utilizzo di questi sistemi in casi in cui i requisiti temporali siano stringenti.

2.2 Feature desiderabili di un sistema real-time

La predicibilità può essere ottenuta apportando pesanti modifiche ai tipici sistemi a time sharing. Alcune proprietà importanti sono:

- **Timeliness**
  I risultati non devono essere corretti solo nel loro valore ma anche nel dominio temporale. Di conseguenza il sistema operativa deve fornire degli specifici meccanismi del kernel per la gestione del tempo e per la gestione dei task con constraint temporali espliciti e criticità diverse.

- **Predicibilità**
  Per raggiungere il livello di performance desiderato, il sistema deve essere analizzabile per prevedere le conseguenze di ogni decisione di scheduling. Nelle applicazioni safety critical tutti i requisiti temporali devono esser garantiti a priori, prima di mettere il sistema in azione. Se qualche vincolo non può essere garantito, il sistema deve essere in grado di informare l'utente e delle decisioni alternative devono essere prese.

- **Efficienza**
  La maggior parte dei sistemi real-time sono embedded, dunque l'efficienza è un requisito fondamentale.

- **Robustezza**
  Il sistema deve essere in grado di continuare a funzionare a qualsiasi condizioni di carico. La gestione dell'overload e il comportamento di adattamento sono feature essenziali per sistemi con molta varianza di carico.

- **Tolleranza agli errori**
  Singoli errori hardware o software non devono portare a un fallimento del sistema.

- **Manutenibilità**
  L'architettura di un sistema real-time deve essere progettata in maniera modulare per permettere di eseguire modifiche facilmente.

2.3 Ottenere la predicibilità

Un sistema dovrebbe essere in grado di prevedere l'evoluzione delle task e garantire in anticipo che tutti i vincoli temporali critici vengano rispettati. L'affidabilità della garanzia dipende da un insieme di fattori, che includono anche le caratteristiche fisiche dell'hardware e i meccanismi/politiche del kernel.
La prima componente che influisce sulla predicibilità è il processore. Le caratteristiche del processore, ad esempio il prefetch delle istruzioni, pipelining, cache, DMA sono fonti di **non determinismo**.
Pur aumentando le performance medie del sistema, introducono fattori di non determinismo che prevengono una precisa stima dei worst-case-execution-time (WCETs). Altre caratteristiche che influenzano la predicibilità sono le caratteristiche interne del kernel,

come l'algoritmo di scheduling, i meccanismi di sincronizzazione, i tipi di semafori, le politiche di gestione della memoria e le politiche di gestione degli interrupt.

2.4 DMA

Uno dei modi più comuni per implementarlo è tramite cycle stealing, il DMA controller prende il controllo del bus per un ciclo di clock e poi lo rilascia al processore. Durante l'operazione di DMA il trasferimento e l'esecuzione del codice sul processore avvengo in parallelo. SE la CPU e il DMA controller richiedono un ciclo di memoria allo stesso tempo, il bus viene assegnato al DMA controller e la CPU aspetta finché il DMA non ha finito. In questo modo non è possibile prevedere il tempo di esecuzione di un task, dato che il tempo di esecuzione dipende da quante volte il DMA controller ha bisogno del bus. Una possibile soluzione al problema è il time slicing, dove un ciclo di memoria è diviso in due slot di tempo adiacenti, uno per il DMA e uno per la CPU. Questa soluzione è più costosa ma più prevedibile di cycle stealing, dato che il response time di una task non è influenzato dal DMA.

2.5 Cache

Nei sistemi real-time la cache è una fonte di non determinismo, dato che il tempo di accesso alla memoria dipende dalla presenza o meno del dato in cache. Quando avviene un cache miss il tempo di accesso alla memoria è molto più lungo, e questo non è prevedibile. Quando si effettua una scrittura in memoria l'utilizzo della cache risulta costoso perché il tempo di accesso alla memoria è più lungo, dato che deve essere garantita la consistenza.

2.6 Interrupts

Un grosso problema per la predicibilità è la gestione degli interrupt provenienti dalle periferiche I/O. Se non gestite correttamente possono introdurre ritardi non limitati durante l'esecuzione di processi.
In quasi tutti i sistemi operativi la gestione di un interrupt prevede l'esecuzione di una service routine dedicata per la gestione del device. Il vantaggio di questo metodo è che tutti i dettagli hardware sono contenuti nel driver.
In molti OS gli interrupt vengono serviti usando un sistema a priorità fissa, tramite i quali ogni driver viene schedulato in base a una priorità statica, questa priorità è più alta rispetto a quella di qualsiasi processo. In generale è molto difficile prevedere a priori il numero di interrupt che un task può ricevere, dunque il delay è impredicibile.
Per ridurre l'interferenza dei driver sui task ed effettuare operazioni di I/O le periferiche devono essere gestiste differentemente

2.6.1 Approccio A

La soluzione più radicale è disattivare tutti gli interrupt esterni, a eccezione di quelli generati dal timer. Tutte le periferiche vanno gestiste dall'applicazione (controllo di programma), i quali hanno accesso ai registri delle periferiche. Dato che non ci sono più interrupt, il trasferimento deve essere effettuato tramite polling. L'accesso diretto alle periferiche consente alta flessibilità al livello di programma ed elimina i delay introdotti dai driver. Dunque il tempo necessario per un trasferimento può essere calcolato a priori. Un altro vantaggio è che il kernel non va modificato quando una nuova periferica viene

aggiunta o ne viene rimossa una.
Lo svantaggio principale è che risulta poco efficiente dato la busy wait necessaria per il polling. Un grosso problema è che le applicazioni richiedono la conoscenza completa dei dettagli a basso livello delle periferiche che devono gestire. Questo può essere gestito incapsulando le routine che dipendono dai device in una libreria.

2.6.2 Approccio B

Come nella soluzione precedente tutti gli interrupt esterni sono disabilitati, tranne quelli dei timer. In questa soluzione le periferiche non vengono gestite direttamente dall'applicazione, ma a turno da routine dedicate del kernel, periodicamente attivate dal timer. Questo approccio elimina i ritardi causati dai driver e confina le operazioni di I/O a una o più routine periodiche del kernel. In alcuni sistemi i device I/O vengono suddivisi in due gruppi in base alla loro velocità, quelli più veloci vengono gestiti da task periodici con frequenza maggiore.
Il vantaggio di questo metodo è che tutti i dettagli hardware sono incapsulati nel kernel, dunque l'applicazione non ha bisogno di conoscere i dettagli a basso livello delle periferiche. Dato che gli interrupt sono disabilitati il maggior problema è dato dalla busy wait delle procedure di gestione dell'I/O del kernel. Questo metodo introduce un overhead maggiore rispetto all'approccio A, richiede modifiche al kernel quando una periferica viene aggiunta o rimossa.

2.6.3 Approccio C

Lasciare gli interrupt abilitati e ridurre al minimo possibile i driver per le periferiche. L'unico scopo dei driver è quello di attivare una task che gestisce il device. Una volta attivato il task di gestione del device viene eseguito sotto diretto controllo del sistema operativo ed è schedulato come un task normale. In questo modo la priorità che può essere assegnata al task di gestione del device è completamente indipendente dalle altre priorità può essere impostata in base ai requisiti del programma.
Il vantaggio principale di questo metodo è che il busy wait viene eliminato e il tempo

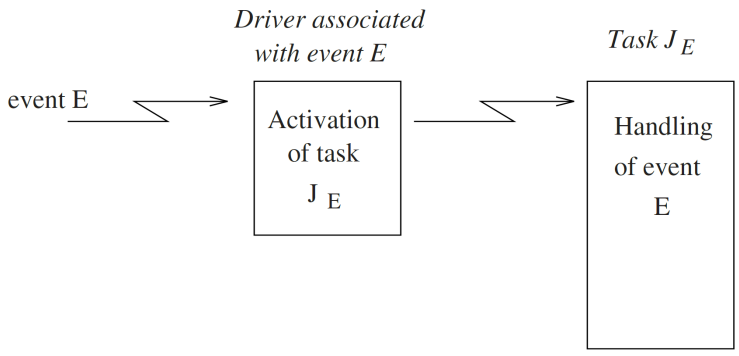

Figure 1: Attivazione task per la gestione del device

di esecuzione del driver di gestione del device è molto ridotto, di conseguenza il tempo di esecuzione del task è più prevedibile. Nella maggior parte delle applicazioni pratiche il piccolo overhead portato da questo tipo di driver viene ignorato.