Problematiche di sistema

RTS

Anno accademico 2008/9
Sistemi *Real-Time*

Tullio Vardanega, tullio.vardanega@math.unipd.it

Context switch

- Il prerilascio ha un costo temporale non nullo che è saggio includere nei *test* di *schedulability*
- In un sistema con prerilascio ogni *job* subisce almeno due *context swicth*
  - Uno all'avvio di esecuzione e uno al suo completamento
- Tale costo va computato a carico del *job*
  - Non è difficile raffinare questo tipo di analisi quando si conoscono le caratteristiche dell'ambiente d'esecuzione

Livelli di priorità – 1

- Le tecniche di analisi FPS viste finora assumono che i *job* abbiano priorità distinte
- Non è però sempre possibile garantire questa condizione
  - Più *job* possono dover condividere la medesima priorità
  - Allo stesso livello di priorità la selezione può avvenire su base FIFO oppure *round-robin*
- Ciò configura il caso peggiore da contemplare in analisi
  - Che il *job* J_i sia rilasciato immediatamente dopo tutti gli altri *job* compresenti allo stesso livello di priorità

Livelli di priorità – 2

- Se $T_\varepsilon(i)$ denota l'insieme di *job* con priorità uguale a J_i l'equazione di *time demand* per J_i diventa
- $W_{i,1}(t) = e_i + b_i + \Sigma_{j \in T_\varepsilon(i)} e_j + \Sigma_{(k=1,\ldots,i-1)} \lceil t/p_k \rceil e_k$
  - Da studiare nell'intervallo $0 < t \leq \min(D_i, p_i)$
- Con chiaro peggioramento del *response time* di J_i
  - L'effetto a livello sistema in termini di *schedulability loss* può però essere meno pesante

Livelli di priorità – 3

- Quando il numero Ω_n di *assigned priorities* è maggiore del numero Ω_s di priorità disponibili serve un *mapping* da Ω_n a Ω_s
  - In generale tutte le *assigned priorities* $\geq \pi_1$ assumono valore π_1 e tutte quelle nell'intervallo $(\pi_{k-1}, \pi_k]$ valore π_k a decrescere nell'intervallo $1 < k \leq \Omega_s$
- Due tecniche
  - *Uniform mapping*
  - *Constant ratio mapping* [Lehoczky & Sha, 1986]

Livelli di priorità – 4

- *Uniform mapping*
  - Disponibilità distribuita uniformemente sulle necessità
  - $Q = \lfloor \Omega_n / \Omega_s \rfloor \Rightarrow \pi_k = kQ$ per $k=1,2,\ldots, \Omega_s-1$ e $\pi_{\Omega s} = \Omega_n$
  - Esempio: con $\Omega_n=10$ e $\Omega_s=3$ da cui $\pi_1=3$, $\pi_2=6$, $\pi_3=10$ per cui avremo 1-3 $\rightarrow \pi_1$, 4-6 $\rightarrow \pi_2$, 7-10 $\rightarrow \pi_3$
- *Constant ratio mapping*
  - Cerca di mantenere costante il rapporto $(\pi_{i-1}+1)/\pi_i$ per $i=2,\ldots, \Omega_s$ per la miglior convenienza dei *job* a priorità maggiore
  - Esempio (come sopra): $\pi_1=1$, $\pi_2=4$, $\pi_3=10$ con *ratio* ½ per cui avremo 1 $\rightarrow \pi_1$, 2-4 $\rightarrow \pi_2$, 5-10 $\rightarrow \pi_3$

Problematiche di sistema

Livelli di priorità – 5

- Lehoczky e Sha hanno mostrato che con *constant ratio mapping* la *schedulable utilization* dell'algoritmo RM deteriora accettabilmente
  - Per g valore minimo di *ratio* nell'intervallo dato il limite ln 2 originario diventa f(g)=ln(2g)+1-g per g>1/2 oppure f(g)=g se g≤1/2
  - Il rapporto f(g)/ln 2 viene detto *relative schedulability*
  - Esempio: con Ω_s = 256 e Ω_n = 100.000 la *relative schedulability* vale 0,9986
    - Sotto regime RMS 256 priorità sono dunque sufficienti

Problematiche di sistema

Tick scheduling – 1

- L'assunzione implicita nelle equazioni viste finora è che lo *scheduler* sia *event-driven*
  - Lo *scheduler* esegue al verificarsi di uno "*scheduling event*" (punto di decisione)
  - Per questo possiamo assumere che un *job* venga inserito nella coda *ready* quando diventa *ready*
- Assunzione non sempre valida: lo *scheduler* talvolta è realizzato in modo *time-driven*
  - Le decisioni di *scheduling* sono prese e attuate periodicamente a intervalli detti *clock interrupt*
  - Questa modalità di *scheduling* viene detta *tick scheduling*

Problematiche di sistema

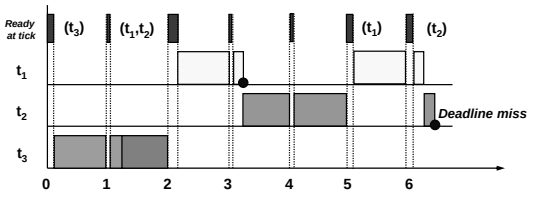
Tick scheduling – 2

- Con *tick scheduling* il momento in cui lo *scheduler* si accorge di un *job* divenuto *ready* può ritardare di un *clock interrupt*
  - Questo ritardo può avere forti ripercussioni sul *response time* del *job*
  - Inoltre deve esistere previsto nel sistema un luogo logico dove trattenere i *job* in questa condizione
    - Il tempo di trasferimento dei *job* da questo luogo alla coda *ready* non è zero e va computato nello *schedulability test*
    - Insieme al costo temporale di gestire il *clock interrupt* in quanto tale

Problematiche di sistema

Esempio

T = {t_3=(0.1,4,1,4), t_2=(0.1,5,1.8,5), t_3=(0,20,5,20)}
con la prima sezione di t_3 non prerilasciabile e ampia 1.1
Con RTA e *scheduler event-driven* si ha R_1=2.1, R_2=3.9, R_3=14.4 (OK)
ma con *tick scheduling* a periodo 1 e costo 0.05 + (0.06 * n) ?



Problematiche di sistema

Tick scheduling – 3

- L'effetto del *tick scheduling* può essere incorporato nella RTA del *job* J_i
  - Introducendo un *task* fittizio $T_0 = (p_0, e_0)$ nel livello di priorità massima con e_0 costo del servizio di *interrupt*
  - Computando il costo m_0 di porre in coda *ready* tutti i *job* J_k a priorità uguale o maggiore di J_i (considerando anche tutte le loro eventuali auto-sospensioni)
    - Il costo corrispondente viene aggiunto al valore e_k
  - Rappresentando il costo di porre in coda *ready* i *job* J_l a priorità inferiore di J_i come task fittizio (p_l, m_0) per ogni l
  - Calcolando il fattore $b_i(np)$ come funzione di p_0
    - $b_i(np) = (\lceil \max_k \theta_k / p_0 \rceil + 1)\, p_0$

Problematiche di sistema

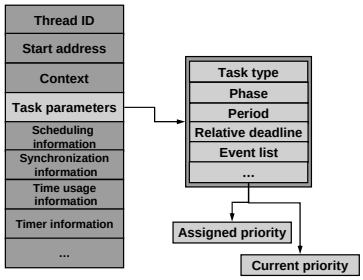
Il sistema operativo – 1

- Ristretto, modulare, estensibile
  - Ristretto (= *small footprint*) perché spesso risiedono in ROM e il sistema ha poca RAM a disposizione
  - Modulare perché deve facilitarne verifica e validazione e anche la valutazione di predicibilità temporale
  - Estensibile perché quale sistema può richiedere funzionalità aggiuntive oltre a quelle minime necessarie
- Ispirato al principio architetturale del *microkernel*
  - I servizi minimi includono *scheduling*, comunicazione e sincronizzazione, gestione delle interruzioni e opzionalmente un minimo di gestione della memoria principale

Il sistema operativo – 2

- I *thread* sono noti al sistema operativo
  - L'unità di allocazione della CPU da parte dello *scheduler*
    - I *thread* emettono i *job* che sono l'oggetto dello *scheduling*
  - La creazione di un *thread* comporta assegnazione di memoria per il suo *Thread Control Block*
  - L'inserimento di un *thread* in una coda di stato (p.es., *ready*) avviene per inserzione di un puntatore al suo TCB
  - La distruzione di un *thread* a fine vita comporta la rimozione del suo TCB eventualmente deallocando dalla memoria il codice e i dati del *thread*
    - Nelle applicazioni *embedded* i *task* che emettono *thread* sono spesso infiniti

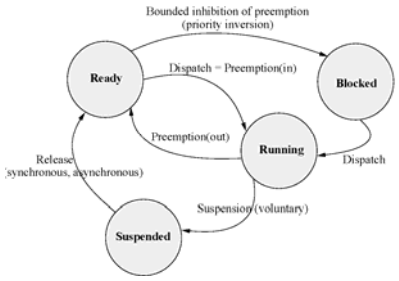
Thread control block



Il sistema operativo – 3

- Tipicamente i *thread* sono realizzati a livello di applicazione invece che come entità primitive di sistema operativo
  - *Thread* periodico
    - Un *thread* che ripetitivamente esegue il codice del *job* fino a un punto di sospensione
  - *Thread* sporadico
    - Un *thread* che emette un *job* in risposta a un evento, ed esegue il codice del *job* fino a un punto di attesa di evento
  - *Task* aperiodico
    - Indistinguibile dagli altri se non per l'assenza di *deadline*

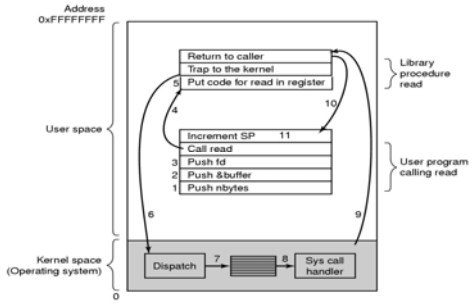
Stati di *thread* – 1



Stati di *thread* – 2

- Nello stato *Suspended* si entra per sospensione volontaria
  - Sospensione periodica per *task* periodico
  - Sospensione sporadica per *task* sporadico
- Servono strutture di gestione diverse
  - Una coda di sospensione temporale per i *thread* di *task* periodici
  - Strutture di attesa di eventi per i *thread* di *task* sporadici

Chiamate di sistema – 1



Chiamate di sistema – 2

- La maggior parte dei servizi del sistema operativo sono eseguiti in risposta a invocazioni esplicite di processi
  - Chiamate di sistema
- Le chiamate di sistema sono nascoste in procedure di libreria predefinite note ai compilatori
  - Ciò vale anche per i sistemi *embedded*
  - L'applicazione non effettua direttamente chiamate di sistema
  - La procedura di libreria svolge il lavoro di preparazione necessario ad assicurare la corretta invocazione della chiamata di sistema
- Sistema operativo e applicazione condividono memoria
  - Contrariamente al caso dei sistemi operativi *general-purpose*
  - Per fiducia nell'applicazione e per efficienza spaziale e temporale
  - Ma allora il sistema operativo deve proteggere le sue strutture di controllo dal rischio di *race condition*

Lo *scheduler*

- Deve eseguire a ogni cambio di stato di *thread*
  - Punti di decisione (*dispatching point*)
- Attivazione tipicamente periodica a seguito di *clock interrupt*
- A ogni *clock interrupt* lo *scheduler* deve
  - Gestire la coda degli eventi temporali accumulatisi dall'ultima ispezione
  - Far avanzare il contatore di consumo temporale del *thread* in esecuzione (per politica *round-robin* ma non solo)
  - Aggiornare e valutare la coda *ready*
- Il *tick size* tipico nell'ordine di 10 ms non è adatto ai sistemi *real-time*
  - Ma una frequenza maggiore può causare *overhead* eccessivo
  - Allo *scheduler* serve dunque capacità di esecuzione *event-driven*

Gestione delle interruzioni – 1

- Le interruzioni *hardware* sono il modo più efficace per notificare l'applicazione di eventi esterni e del completamento di attività di I/O asincrone
- Le attività di gestione variano per frequenza e intensità secondo la sorgente dell'interruzione
- Il servizio di gestione è perciò spesso realizzato in due fasi distinte (immediata e differita)
  - Quella immediata si svolge al livello di priorità delle interruzioni superiore alle priorità *software*
  - Quella differita come attività *software*

Gestione delle interruzioni – 2

- Alla notifica *hardware* di una interruzione il processore salva i registri PC e PSW nello *stack* delle interruzioni e salta all'indirizzo dell'ISR selezionata
  - In questa fase le interruzioni vengono disabilitate per evitare rischi di *race condition* a seguito di nuove notifiche
- Svolte le azioni di gestione immediata il processore ripristina i registri e riabilita le interruzioni
  - La sorgente di interruzione a un dato livello di priorità può essere localizzata tramite *polling* oppure *interrupt vector*
    - Il *polling* è piu generale rispetto alle caratteristiche dei dispositivi
    - Il *vectoring* facilita la localizzazione del servizio richiesto

Gestione delle interruzioni – 3

- Il servizio di gestione delle interruzioni ha latenza massima determinata dal tempo necessario per
  - Completare l'istruzione corrente, salvare i registri, pulire la pipeline, acquisire il vettore delle interruzioni, attivare il meccanismo di *trap*
  - Disabilitare le interruzioni
  - Completare l'esecuzione di ISR a priorità maggiore
    - Fattore di interferenza a livello interruzioni
  - Salvare il contesto del *thread* interrotto, identificare la sorgente di interruzione, saltare al suo codice
  - Avviare l'esecuzione dell'ISR
    - Che può avere una parte *device-independent* e una *device-specific*

Gestione delle interruzioni – 4

- La parte differita deve essere prerilasciabile dal resto dell'applicazione
  - Dunque esegue con priorità a livello *software*
- Può operare su strutture dati di *kernel*
  - Conviene che siano incapsulate in sezioni critiche protette da adeguati protocolli di accesso
  - In questo modo non serve avere *task* di livello *kernel*
- Per avere migliore *responsiveness* alle interruzioni conviene usare *slack stealing* o altri schemi *bandwidth preserving*
  - Ma per realizzarli al meglio serve supporto dal sistema operativo

Gestione del tempo – 1

- **Un orologio di sistema consiste di**
  - **Un registro contatore con periodo**
    - Automaticamente re-inizializzato al valore del periodo (*tick size*) ogni volta che raggiunge il *triggering edge* e produce il *clock tick*
    - Ha una parte *hardware* aggiornata a ogni oscillazione e una parte *software* aggiornata a ogni gestione di *clock tick*
  - **Una coda di attesa di eventi temporali**
  - **Un gestore delle interruzioni (prevalentemente immediato)**
- **La frequenza del *clock tick* fissa la risoluzione dell'orologio *software***
  - **Conviene che la risoluzione divida il *tick size***
    - In modo che il *kernel* faccia *tick scheduling* ogni N *tick* del *clock*
    - Le interruzioni frequenti sono dette *time-service interrupt* e le multiple *clock interrupt*

Gestione del tempo – 2

- **La risoluzione dell'orologio *software* è un parametro di progetto del *kernel***
  - **Più fina la risoluzione maggiore l'*overhead* di gestione dei *time-service interrupt***
- **C'è un rapporto delicato tra l'accuratezza richiesta dall'applicazione e quella ottenibile**
  - **L'interrogazione dell'orologio *software* da un *task* di applicazione soffre necessariamente di latenza**
  - **La risoluzione richiesta non può essere più fina della latenza massima**
    - Nell'ordine di un centinaio di microsecondi

Gestione del tempo – 3

- **Oltre agli orologi periodici vi sono quelli *one-shot***
  - **Operanti in modalità programmata non ripetitiva**
    - *Interval timer*
- **Il *kernel* scandisce la coda degli eventi attesi per determinare il tempo della prossima interruzione**
  - **La risoluzione di questo orologio è limitata dal tempo necessario al *kernel* per farne gestione**
    - Oggi nell'ordine di microsecondi

Gestione del tempo – 4

- **L'accuratezza degli eventi temporali è misurata come la differenza tra il tempo richiesto e quello ottenuto**
- **Dipende da tre fattori**
  - **La frequenza con la quale le code degli eventi sono ispezionate**
    - Corrispondente al periodo dei *time-service interrupt* a meno di usare *interval timer*
  - **La politica con la quale il *kernel* gestisce la coda degli eventi**
    - LIFO vs. FIFO
  - **Il costo di gestione della coda degli eventi**
- **Ne segue che il *release time* dei *task* periodici soffre di *jitter***