Problematiche di ordinamento

RTS

Anno accademico 2008/9
Sistemi *Real-Time*

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

Prima classificazione – 1

- **Funzionamento *clock-driven* (*time-driven*)**
  - **Le decisioni di ordinamento sono prese prima dell'esecuzione (*off line*) e sono poi attuate a istanti temporali predeterminati**
    - In genere a intervalli regolari segnalati da una interruzione di orologio
    - Lo *scheduler* prima avvia all'esecuzione i *job* pianificati nel periodo corrente poi si sospende in attesa del prossimo periodo
    - Lo *scheduler* usa un *off-line schedule*
  - **Tutti i parametri di influenza devono essere noti a priori**
  - **L'ordinamento non è modificabile a tempo d'esecuzione**
  - **L'onere di gestione a tempo d'esecuzione è minimo**

Prima classificazione – 2

- **Funzionamento *weighted round-robin***
  - ***Round-robin* base**
    - I *job* pronti vengono inseriti in una coda FIFO
    - Il *job* in testa alla coda esegue per un *time slice*
      - Se non finisce in tempo viene prerilasciato e posto in fondo alla coda
    - La coda viene eseguita in un *round*
  - **Correzione con pesi**
    - Usato per la gestione del traffico di rete
    - Ai *job* vengono assegnate quantità diverse di tempo di CPU secondo il valore dell'attributo 'peso'
    - Totalmente inadatto per *job* con relazioni di precedenza
    - Adatto per *job* operanti in modalità *pipeline*

Prima classificazione – 3

- **Funzionamento *priority-driven* (*event-driven*)**
  - **Questa classe di algoritmi non lascia mai risorse inutilizzate intenzionalmente**
    - Una risorsa non viene utilizzata solo e soltanto quando non vi sia alcun *job* pronto a usarla (approccio *greedy* che cerca l'ottimo locale)
    - Ritardare l'accesso alla CPU può però ridurre la competizione e dunque il ritardo nel completamento dei *job*
    - Effetti anomali quando i parametri dei *job* possano variare dinamicamente
  - **Le decisioni di ordinamento vengono prese quando si verificano eventi che modificano la coda dei pronti**
    - Algoritmi tipicamente *on-line* e solo con conoscenza locale
    - Uno *scheduler* che conosca il tempo di arrivo futuro dei suoi *job* viene detto *clairvoyant*
  - **Include algoritmi di *scheduling* utilizzati in sistemi non *real-time* come FIFO, LIFO, SETF, LEFT**

Prima classificazione – 4

- **Come confrontare le prestazioni dell'ordinamento *preemptive* con quello *non-preemptive*?**
  - **Non c'è risposta valida in generale**
    - Nel caso particolare in cui tutti i *job* considerati abbiano lo stesso *release time* la modalità *preemptive* è migliore se il costo del prerilascio è insignificante
  - **Sarebbe interessante sapere se il miglioramento del *minimum makespan* nel caso *preemptive* bilanci in generale il costo del prerilascio**
    - Sappiamo solo che nel caso di 2 processori il *mimimum makespan* nel caso *non-preemptive* non è mai peggiore di 4/3 di quello del caso *preemptive*

Definizioni

- **I vincoli di precedenza hanno influenza su *release time* e *deadline***
  - **Altrimenti il *release time* di un *job* potrebbe essere dopo quello di un suo successore o la sua *deadline* prima di quella di un suo predecessore**
- ***Effective release time***
  - **Per un *job* con predecessori è il massimo valore tra il suo *release time* e l'*effective release time* dei suoi predecessori**
    - Più precisamente il massimo *effective release time* dei suoi predecessori più il tempo d'esecuzione del *job* corrispondente
- ***Effective deadline***
  - **Per un *job* con successori è il minimo valore tra la sua *deadline* e la *effective deadline* dei suoi successori**
    - Più precisamente la minima *effective deadline* dei suoi successori meno il tempo d'esecuzione del job corrispondente
- **Nel caso di 1 processore e *job* con prerilascio, considerando ERT e ED possiamo trascurare i vincoli di precedenza**

Ottimalità – 1

- **Le priorità possono essere assegnate sulla base della (*effective*) *deadline***
  - **L'algoritmo *Earliest Deadline First* è ottimale per sistemi monoprocessore con prerilascio e *job* indipendenti**
    - Dato un insieme di *job* EDF produce un *feasible schedule* se uno ne esiste
    - L'ottimalità cade sotto ipotesi diverse (assenza di prerilascio o più processori)
- **La priorità può anche essere assegnata sulla base dello *slack* (*laxity*)**
  - **Lo *slack* al tempo t di un job con *deadline* d e tempo residuo di esecuzione r è [(d – t) – r]**
  - **L'algoritmo *Least Slack Time First* (*Least Laxity First*) è ottimale sotto le stesse ipotesi di EDF**
    - Però è più complesso di EDF perché richiede conoscenza del tempo d'esecuzione

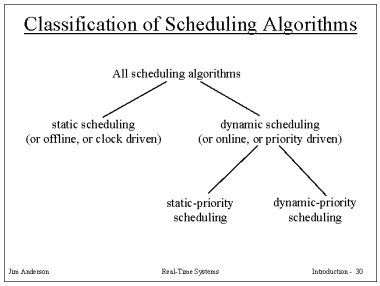
Ottimalità – 2

- **Se l'obiettivo è rispettare le scadenze non c'è vero vantaggio nel far completare un *job* prima del necessario**
  - **Da questa osservazione nasce l'algoritmo *Latest Release Time* che ordina i *job* all'indietro a partire dalle loro *deadline***
    - Per primo si fissa il *job* con la *deadline* più lontana e poi, all'indietro, i *job* con il maggior *release time* (considerato come priorità)
    - LRT non è *priority-driven* perché può ritardare l'esecuzione di un *ready job*
  - **Il punto chiave è che la natura *greedy* degli algoritmi *priority-driven* può aumentare l'interferenza**

Predicibilità di esecuzione

- **Definizione intuitiva**
  - **Da un algoritmo di *scheduling* l'esecuzione del *job set* J è *predictable* se l'*actual start time* e l'*actual response time* di ogni suo *job* variano entro i corrispondenti valori fissati dal suo *maximal* e *minimal schedule***
    - Lo *schedule* prodotto dall'algoritmo dato sotto ipotesi WCET per ogni *job* viene detto *maximal schedule*
    - Analogamente per *minimal schedule*
- **L'esecuzione con prerilascio di job indipendenti e con *release time* fissati sotto *priority-driven scheduling* su 1 processore è predicibile**

Prima classificazione – 5



Clock-driven scheduling – 1

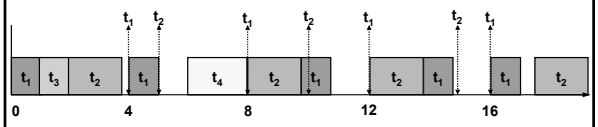
- ***Workload model* composto da**
  - **N task periodici per N fissato**
    - Nella definizione di periodico di Jim Anderson (rispetto a quella di Jane Liu)
  - **I parametri di ogni *task* T_i (Φ_i, p_i, e_i, D_i) sono noti a priori**
- **Lo *schedule* è statico, costruito *off-line* (prima dell'esecuzione dei *job*) in una tabella T**
  - **$T[t_k] = T_i$ se un *job* deve essere mandato in esecuzione al tempo t_k altrimenti I (*idle*)**

Clock-driven scheduling – 2

```
Input: stored schedule T(t_k) for k = 0,..,N-1; H (hyperperiod)
SCHEDULER:
   i := 0; T(t_k) = 0;
   set the timer to expire at t_k;
   do forever:
      accept timer interrupt;
      if an aperiodic job is executing
         preempt it; end if;
      current task T := T(t_k);
      i := i+1;
      compute T(t_k) for k = i mod N;
      set the timer to expire at ⌊ i / N ⌋ H + t_k;
      if current task T = I
        let the job at the head of the aperiodic queue execute;
      else let the job of task T execute; end if;
      sleep;
   end do;
end SCHEDULER
```

Esempio

T per J = {t_1=(0,4,1,4), t_2=(0,5,1.8,5), t_3=(0,20,1,20), t_4=(0,20,2,20}
H = 20



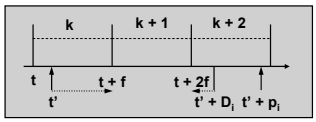
T contiene 17 celle: troppo complessa e troppo frammentata!

Clock-driven scheduling – 3

- **Conviene minimizzare la dimensione di T**
  - **I punti di decisione t_k sono fissati a intervalli regolari**
    - Gli intervalli sono chiamati *minor cycle* e hanno ampiezza f
    - Entro un *minor cycle* non vi è prerilascio
  - **Φ_i per ogni task T_i è un multiplo intero non negativo di f**
    - Il primo *job* di ogni *task* ha il suo *release time* all'inizio di un *minor cycle*
  - **[1] Ogni *job* deve eseguire completamente entro f**
    - $f \geq \max_i (e_i)$ così che si possa riconoscere la situazione di *overrun*
  - **[2] f è un divisore dell'iperperiodo H**
    - L'iperperiodo H contiene un numero intero F di *minor cycle*
    - L'iperperiodo H che inizia all'inizio del *frame* di indice kF per k=0,…,N-1 viene detto *major cycle*

Clock-driven scheduling – 4

- **Ulteriori vincoli**
  - **Per consentire allo *scheduler* di accertare che ogni *job* completi entro la sua *deadline* conviene che tra *release time* e *deadline* di ogni *job* vi sia almeno 1 *minor cycle***



  - **[3] Deve valere la relazione $t + 2f \leq t' + D_i$ da cui il vincolo $2f - GCD(p_i, f) \leq D_i$**

Clock-driven scheduling – 5

- **È possibile che i parametri di qualche sistema non riescano a soddisfare le condizioni [1-3] su f**
- **Esempio**
  - **T = {(0,4,1,4), (0,5,2,7), (0,20,5,20)}**
  - **[1] f ≥ 5; [2] f = {2,4,5,20}; [3] f ≤ 4**
- **In questi casi occorre decomporre i job con e_i "grande" in frammenti (*slice*) piccoli abbastanza da soddisfare i vincoli violati**

Clock-driven scheduling – 6

- **Per costruire un *cyclic schedule* bisogna prendere tre decisioni**
  - **Fissare f**
  - **Suddividere *job* grandi in *slice***
  - **Assegnare *slice* a *minor cycle***
- **Vi è interdipendenza tra queste decisioni**

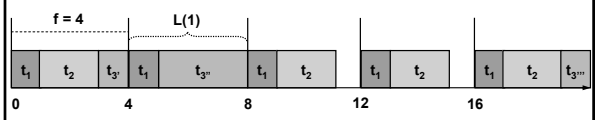
Clock-driven scheduling – 7

```
Input: stored schedule L(k) for k = 0,..,F-1;
CYCLIC_EXECUTIVE:
   t := 0; k = 0;
   do forever:
      accept clock interrupt at time t*f;
      currentBlock = L(k);
      t := t+1;
      k := t mod F;
      if the last job is not completed take appropriate action;
      end if;
      execute the slices in currentBlock;
      while the aperiodic job queue is not empty do
        execute the aperiodic job at the top of the queue;
      end do;
      sleep until the next clock interrupt;
   end do;
end SCHEDULER
```

Problematiche di ordinamento

Esempio

T per J = {t_1=(0,4,1,4), t_2=(0,5,2,7), t_3=(0,20,5,20)}
Abbiamo un problema con t_3 perché vorremmo $e_3 \le f \le 4$
Dunque lo scomponiamo in più *slice*: quanti?



t_3 = {$t_{3'}$=(0,20,1,20), $t_{3''}$=(0,20,3,20), $t_{3'''}$=(0,20,1,20)}

Major cycle H = 20, F = H / f = 5

Problematiche di ordinamento

Considerazioni progettuali – 1

- Completare un *job* in anticipo rispetto alla sua *deadline* non porta beneficio
- Dare opportunità a *job* aperiodici (*event-driven*) al più presto possibile rende il sistema più *responsive*
- *Slack stealing* fa eseguire *job* aperiodici in preferenza a *job* periodici quando possibile
  - Ogni *minor cycle* ha una quota X di tempo non impiegato (*slack*)
    - Lo *slack* è dunque un attributo statico di ogni *minor cycle*
  - Lo *scheduler* fa *slack stealing* se assegna lo *slack* all'inizio del *minor cycle* invece che alla fine
    - Serve un *interval timer* molto accurato per segnalare la fine dello *slack*!

Problematiche di ordinamento

Considerazioni progettuali – 2

- Gestire gli *overrun*
  - Fermare il *job* trovato in *overrun* all'inizio del nuovo *minor cycle*
    - Era davvero sua la colpa?
    - Accettabile nello schema in cui un risultato tardivo ha valore nullo o negativo
  - Ritardare l'arresto a dopo il completamento di azioni critiche
    - L'arresto prematuro può produrre uno stato inconsistente
  - Assegnare tempo supplementare ritardando il nuovo *frame*
    - Accettabile nello schema in cui un risultato tardivo ha valore positivo
- Gestire il cambio di modo operativo (*mode change*)
  - Con o senza *deadline*
  - Sovrapposizione tra il modo precedente con quello successivo

Problematiche di ordinamento

Clock-driven scheduling – 8

- Pro
  - Semplicità concettuale
  - Realizzazione molto semplice e robusta
  - Verifica sicura ed economica
- Contro
  - Progettazione estremamente fragile
    - Costruire la tabella T è problema *NP-hard*
    - Elevatissimo accoppiamento architetturale
  - Tutti i parametri devono essere noti e fissati a priori
    - Spesso fissati in modo arbitrario per soddisfare i vincoli su f
    - Inadatto e inefficiente per *job* sporadici

Problematiche di ordinamento

Priority-driven scheduling

- Principio base
  - A ogni *job* viene assegnata una priorità
  - Il *job* con la priorità più alta va in esecuzione
- *Dynamic-priority scheduling*
  - *Job* diversi di uno stesso *task* possono avere priorità diverse
- *Static-priority scheduling*
  - I *job* dello stesso *task* hanno tutti la stessa priorità

Problematiche di ordinamento

Dynamic-priority scheduling

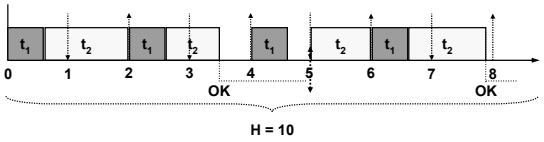
- Due algoritmi principali
  - *Earliest Deadline First* (EDF)
  - *Least Laxity First* (LLF)
- Entrambi ottimali sotto le ipotesi di job indipendenti e con prerilascio
  - Teorema (EDF: Liu & Layland, 1973) valido anche in presenza di *task* sporadici
    - Esteso banalmente a LLF
  - La *deadline* relativa dei *task* periodici può essere arbitraria rispetto al periodo (<, =, >)
  - EDF non è più ottimale per *job* senza prerilascio

Criteri di confronto – 1

- **Gli algoritmi *priority-driven* che non considerano l'urgenza dei *job* hanno prestazioni scadenti**
- **Confrontare le prestazioni di algoritmi diversi**
- **Un buon criterio è la *schedulable utilization***
  - **Un algoritmo può produrre un *feasible schedule* per un *task set* J su 1 processore se U(J) non eccede la sua *schedulable utilization***
  - **La massima *schedulable utilization* teorica è 1**
    - La *schedulable utilization* di EDF è 1 (Liu & Layland, 1973)
  - **Per *deadline* arbitrarie il fattore *density* $\Delta_k = e_k / \min(D_k, p_k)$ diventa importante ($\Delta > U$ se $D_i < p_i$ per qualche *task*)**
    - $\Sigma (e_i/\min(D_i, p_i)) = \Delta \leq 1$ è la *schedulability condition* per EDF: solo sufficiente

Esempio EDF – 1

Task set T = {t_1=(0,2,0.6,1), t_2=(0,5,2.3,5)}
$\Delta(T) = e_1/D_1 + e_2/p_2 = 1.06 > 1$
Che dire di questo T sotto EDF?

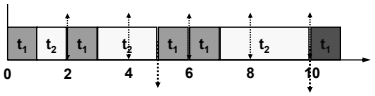


Criteri di confronto – 2

- **La *schedulable utilization* da sola non basta, serve anche la *predictability*!**
  - **Sotto ipotesi di *overload* il comportamento di algoritmi di *static-priority scheduling* è determinabile e ragionevole**
    - L'*overrun* di *job* di un *task* non causa alcun effetto a *task* di priorità superiore
    - L'effetto di un *overrun* sotto EDF è molto più difficile da determinare: EDF causa instabilità
  - **Sotto EDF un *job* che abbia mancato la sua *deadline* ha priorità superiore a *job* con *deadline* nel futuro**
    - EDF causa instabilità

Esempio EDF – 2

Task set T = {t_1=(0,2,1,2), t_2=(0,5,3,5)}
$U(T) = e_1/p_1 + e_2/p_2 = 1.1$
T non ha *feasible schedule*: chi ne paga le conseguenze sotto EDF?



Task set T = {t_1=(0,2,0.8,2), t_2=(0,5,3.5,5)}
$U(T) = e_1/p_1 + e_2/p_2 = 1.1$
T non ha *feasible schedule*: chi ne paga le conseguenze sotto EDF?

E per T = {t1=(0,2,0.8,2), t2=(0,5,4,5)} con U(T)=1.2 invece?

Static-priority scheduling (FPS)

- **Due varianti principali rispetto alla strategia di assegnazione della priorità**
  - ***Rate monotonic***
    - Priorità maggiore a *task* di periodo inferiore (più frequenti)
  - ***Deadline monotonic***
    - Priorità maggiore a *task* con *deadline* inferiore (più urgenti)
- **Prima di studiarli in dettaglio fissiamo alcuni concetti fondamentali**

Critical instant – 1

- **Gli *schedulability test* devono contemplare il caso peggiore**
  - **La peggiore relazione possibile tra il *release time* del *task* T_i da analizzare e i *task* a priorità maggiore di esso**
  - **La sua effettiva specifica può variare a seconda della relazione ammissibile tra D_i e p_i**
- **La nozione di *critical instant* definisce il caso peggiore**
  - **Il *response time* R_i del *job* di un *task* T_i con *release time* nel *critical instant* è il massimo valore possibile per quel *task***

Critical instant – 2

- **Teorema: sotto FPS – quando $D_i \le p_i$ – il *critical instant* del task T_i si ha quando il *release time* di un suo *job* coincide (è in fase) con quello di un *job* di ogni task a priorità maggiore**

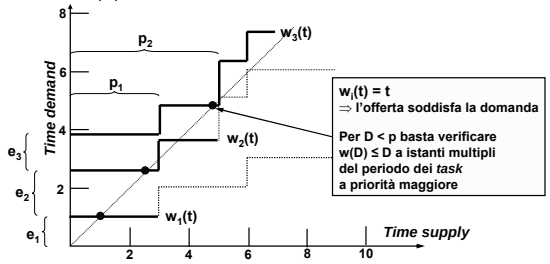
  Si tratta di trovare il valore max(W_i) per T_i

  $W_{i,1} = e_i + \Sigma_{(k=1,..,i-1)} \lceil (W_{i,1} + \Phi_i - \Phi_k)/p_k \rceil e_k - \Phi_i$

- **Concetto alla base della *Time Demand Analysis* che però studia w in funzione di t**

Time demand analysis – 1

Task set T = {t_1=(-,3,1,3), t_2=(-,5,1.5,5), t_3=(-,7,1.25,7)}
U(T) = Σ e_i/p_i = 0.82 e fasi arbitrarie ma ininfluenti nel *critical instant*



Time demand analysis – 2

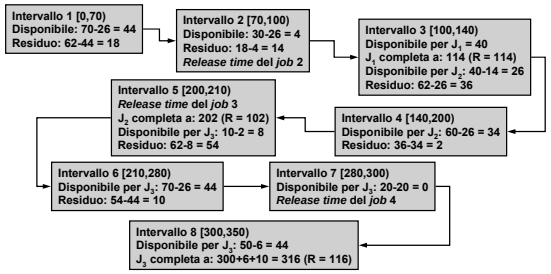
- **La TDA (w(t) ≤ D) è necessaria e sufficiente**
  - **[Lehoczky & Sha & Ding, 1989]**
- **È facile estenderne le potenzialità alla determinazione del *response time***
  - **Risolvendo la funzione iterativa w = f(w)**
  - **[Joseph & Pandia, 1986]**
  - **[Audsley & Burns & Richardson & Tindell & Wellings, 1993]**

Time demand analysis – 3

- **Cambia qualcosa nella definizione di *critical instant* quando D>p ?**
- **Si: il primo *job* di T_i può non avere il peggior *response time***
  - **[Lehoczky & Sha & Strosnider & Tokuda, 1991]**
- **Serve invece considerare tutti i *job* di T_i all'interno del *level-i busy period***
  - **L'intervallo temporale (t_0, t) nel quale il *processor* esegue *job* con priorità ≥i , considerando solo i *job* con *release time* in (t_0, t) e *response time* entro t**

Esempio

T_1 = {-,70,26,70}, T_2 = {-,100,62,120}



Level-i busy period

T_1 = {-,100,20,100}, T_2 = {-,150,40,150}, T3 = {-,350,100,350} ⇒ U = 0.75