

28/09/2022

INTRODUZIONE AI SISTEMI REAL-TIME

Cos'è un sistema real-time?

- È un sistema progettato per operare con vincoli temporali ben definiti:
funziona correttamente se rispetta i vincoli temporali ben definiti per ogni input possibile:
se non lo fa, è come se l'output fornito fosse errato.
- La maggior parte dei sistemi embedded sono anche real-time, e viceversa. Inoltre,
la teoria della schedulazione-real time si preoccupa di rispettare delle scadenze con
a disposizione delle risorse hardware limitate.
- ↳ ESEMPIO DI SISTEMA EMBEDDED E REAL-TIME: airbag, che deve rispettare dei vincoli temporali ben
rigorosi: in caso di vito, l'airbag non può scoppiare né troppo presto né troppo tardi.

Informalmente, molti risultati teorici della teoria della schedulazione real-time assicurano l'aspetto di vincoli temporali di un insieme di processi se l'utilizzazione del dispositivo di calcolo è sotto una certa soglia.

Quando affidiamo le nostre vite a un sistema, vogliamo essere MATEMATICAMENTE CERTI del suo corretto funzionamento: il testing va fatto ma non basta.

Sistemi di controllo ad alto livello:

In genere i sistemi complessi sono controllati da gerarchie di sistemi di controllo.

BASE DI DATI TEMPORALE:

Qui i dati memorizzati hanno un valore tanto maggiore quanto è più recente: se un dato non viene aggiornato da un po' di tempo, non è più buono a nulla.

I sistemi real-time si suddividono in:

- Periodiche calchi
- Periodici calchi
- Asincroni ma grosso modo predibitivi
- Asincroni e impredibitivi.

Definizioni:

→ JOB = unità di lavoro che può essere schedulata ed eseguita in un sistema real-time.

→ TASK = insieme di job correlati che insieme concorrono a realizzare una funzionalità del sistema.

→ PROCESSORE = componente che esegue i job.

→ RISORSA = componente PASSIVA di un sistema real-time la cui presenza è comunque necessaria per l'esecuzione dei job.

N.B.: il processore incide sul tempo di risposta del sistema: al variazione delle sue caratteristiche cambia anche la modellistica / l'analisi da fare. All'interno delle modellistiche, le risorse invece non incidono sul tempo min. di job da eseguire in un'unità di tempo: una risorsa, o c'è o non c'è.

wranio il → TEMPO DI RISPOSTA DI UN JOB = istante di completamento - istante di rilascio,
di cal = dove l'istante di rilascio è l'istante in cui il job diventa disponibile.

→ SCADENZA ASSOLUTA = ~~istante di rilascio~~ istante di rilascio + scadenza relativa.
È un istante di tempo

04/10/2022

Sviluppo dei sistemi embedded:

(risorse finte rispetto a rischi) Fondamentali: ARCHITETTURE

utilizzando una PIATTAFORMA di riferimento modificata per realizzare nuove funzionalità

forma scelta: lo sviluppo viene

RATA (\rightarrow utilizzo di circuiti molto

in circo \Rightarrow per avere una

o componenti commerciali che si possono trovare

area di memoria; sono classificati meno performanti e complessi in un sistema embeddable.

interconnessione completamente faticata.

una certa programmazione, = RIPROGRAMMABILI.

IB: progettare un ASIC da 2020 volte, anche
più complessa.

06/10/2022

Modello di riferimento per i sistemi real-time:

\rightarrow Minimizzare i tempi di risposta di un job in un sistema ^{HARD} real-time è utile e dettero*: è solo necessario che il job rispetti il vincolo temporale prestabilito.

*NA STRATEGIE DETERIORIO: minimizzare i tempi di risposta \equiv aumentarne la varianza.

Pertanto noi vogliamo una certa prevedibilità e costanza nei tempi di risposta.

\rightarrow I sistemi HARD REAL-TIME possono avere vincoli temporali formulati:

- In modo DETERMINISTICO. \rightarrow Nella pratica va quasi sempre così.
- In modo PROBABILISTICO.
- In termini di alcune FUNZIONI DI UTILITÀ.

\rightarrow Nei sistemi SOFT REAL-TIME può essere importante conseguire anche altri obiettivi: come minimizzare il tempo di risposta (a differenza dei sistemi hard-real-time).

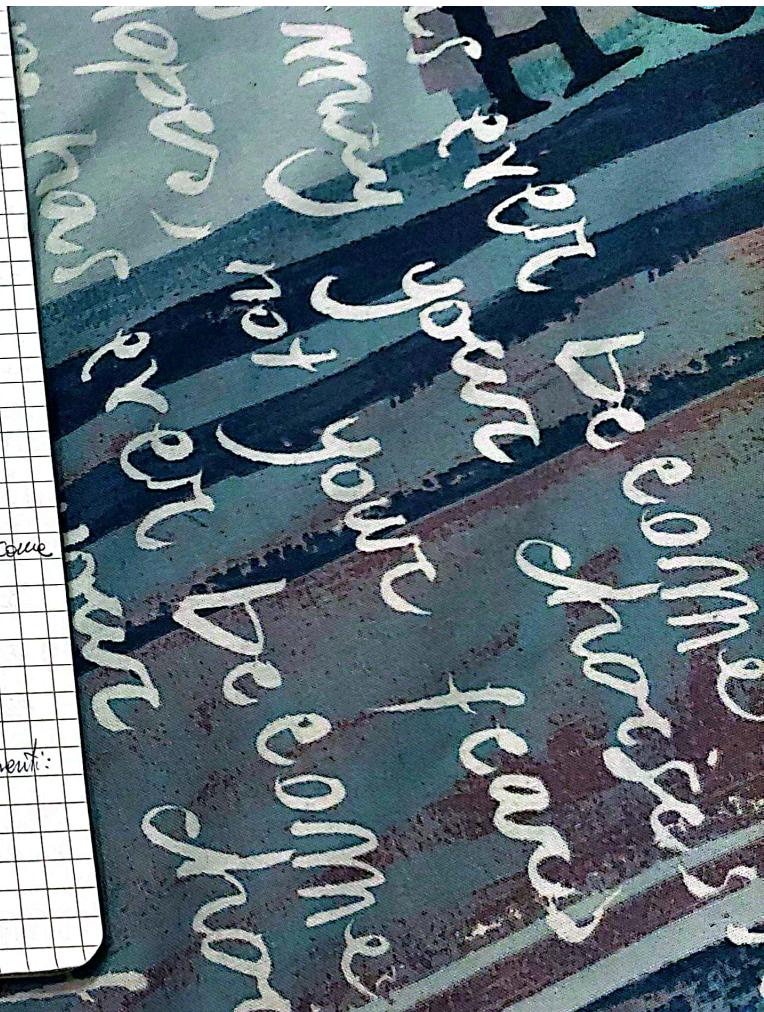
\rightarrow Chiaramente l'utilità del job tardivo dipende dalla specifica applicazione.

Ogni sistema real-time può essere caratterizzato da un modello composto da 3 elementi:

1) Un MODELLO DI CARICO \rightarrow Descrivere le applicazioni supportate dal sistema.

2) Un MODELLO DELLE RISORSE \rightarrow Descrivere le risorse di sistema a disposizione.

3) DGLI ALGORITMI. \rightarrow Definiscono come assegnare le risorse alle applicazioni.



Le RISORSE DI SISTEMA rappresentano soffware. L'ambiente di esecuzione delle applicazioni.
Ne esistono di due tipi: ATTIVE (come i processori) e PASSIVE.

Nel modellare un sistema:

- Si elencano semplicemente i processori come P_1, P_2, \dots, P_m .
- Si elencano semplicemente le risorse come R_1, R_2, \dots, R_p (ma, se è così abbondante da essere sempre disponibile, la risorsa si omette proprio).
- Il carico viene descritto in termini di TASK, ciascuno dei quali è un insieme di job.
- I job avranno dei VINCOLI TEMPORALI, dei PARACETRI FUNZIONALI, delle RISORSE DA USARE, dei PARACETRI DI INTERCONNESSIONE.

VINCOLI TEMPORALI DEI JOBS:

- a) ISTANTE DI RILASCIO (\equiv RELEASE TIME) del job $J_i = r_i$ = istante in cui J_i diventa disponibile.
- b) SCADENZA (\equiv DEADLINE) del job $J_i = d_i$ = istante entro cui J_i deve essere completato.
- c) SCADENZA RELATIVO del job $J_i = D_i = d_i - r_i$.
- d) INTERVALLO DI FATTIBILITÀ = intervallo $(r_i, d_i]$.

→ A volte, le specifiche non determinano esattamente l'istante di rilascio, bensì un intervallo in cui r_i può cadere.

→ Molti sistemi real-time devono poter reagire a eventi esterni che accorrono a istanti casuali.

DEFINIZIONI:

- 1) TASK APERIODICO = task in cui gli istanti di rilascio dei job sono casuali.
- 2) TASK SPORADICO = come il task aperiodico ma l'intervallo di tempo minimo fra il rilascio di due job.
- 3) TEMPO DI ESECUZIONE t_i del job J_i = tempo richiesto per a J_i per completare l'esecuzione nel caso in cui J_i fosse det solo nel sistema.

DIFICOLTÀ: α^3

Stiamo costretti a definire il tempo di esecuzione \leftarrow → Il hardware è talmente complesso da non essere più determinabile tramite un intervallo. \rightarrow Il tempo effettivo varia da esec. a esec.

Talvolta si ricorre alla definizione del WORST CASE EXECUTION TIME (WCET).

Tuttavia, anche definire il WCET è difficile (non va molto bene eredetare da ordini di grandezza).

Il WCET deve essere conosciuto per i job che girano in un sistema hard-real-time.
Quello che si fa è aumentare la predicitività delle pi. dei processori a discapito delle pre-
stazioni; si possono anche prendere provvedimenti a livello software, ad esempio scrivendo program-
mi senza uso di strutture dati dinamiche, o evitando interazioni troppo complesse tra i job, mi-
mizzando il numero di primitive di sincronizzazione delle risorse condivise.

IN TAL MODO SI RENDE IL SISTEMA SPERIMENTALMENTE / QUASI DETERMINISTICO.

MODELLO A TASK PERIODICI:

È un modello di carico deterministico molto conosciuto.

Qui viene definito il periodo p_i del task T_i , che è l'esatto intervallo temporale fra gli istan-
ti di rilascio dei job di T_i . Il tempo di esecuzione e_i , invece, è il max tra tutti i
tempi di esecuzione dei job nel task T_i .

N.B.: Un job non può essere schedulato finché quello precedente non è terminato.

NOTAZIONE:

→ TASK: T_1, T_2, \dots, T_n

→ JOB DEL TASK T_i : J_{i1}, J_{i2}, \dots

→ Cumulativam. tutti i job del sistema sono J_1, J_2, \dots

→ STANTE DI RILASCO $\tau_{ij} \equiv$ FASE DI T_i ($\phi_i = \tau_{ij}$) → SE I TASK SONO IN FASE IL PRIMO RILASCO
IN TUTTI I JOBS AVVIENE NELLO STESSO MOMENTO.

→ $H = \min_i$ di tutti i periodi p_i dei task del sistema. Un intervallo temporale di lung-
ghezza H è chiamato **IPERPERIODO**

→ $N = \sum_{i=1}^n H/p_i = \#$ massimo di job in un iperperiodo.

→ UTILIZZAZIONE DEL TASK $T_i = m_i = e_i/p_i$ (N.B.: può anche essere > 1)

→ UTILIZZAZIONE TOTALE = $U = \sum_{i=1}^n m_i$

→ Tutti i job di uno stesso task T_i hanno una scadenza relativa D_i . In molti casi
(ma non in tutti), possiamo dire che $D_i = p_i$. SCADENZA IMPLICATA

VINCOLI DI PRECEDENZA:

Imponevano un certo ordine fra i job. In generale, complicano il compito di trovare una

La schedulazione per i job, a meno che non si tratti di un ordinamento totale ha i job stessi.

Le precedenze possono essere rappresentate con un GRAFO DI PRECEDENZA.

SCHEDULER:

Sistema di gestione dei job: è il modulo del sistema real-time che implementa algoritmi per ordinare l'esecuzione dei job e controllare l'accesso alle risorse.

↳ Una schedularazione è FATTIBILE (FEASIBLE) se ogni job è completato entro la sua durata.

MISURE DI PRESTAZIONI:

→ TARDIVITÀ: è zero solo se la scadenza è rispettata.

→ LATENESS: differenza tra l'istante di completamento e la scadenza.

→ TEMPO DI RISPOSTA: $t_{istante}$ di Compatimento - $t_{istante}$ di rilascio

MISS RATE : % DI JOB CHE TERMINANO OLTRE LA SCADENZA.

→ LOSS RATE: % JOBS NON ESEGUITI.

PARAMETRI FUNZIONALI:

→ INTERRUPIBILITÀ (\equiv preemption).

→ CRITICALITÀ (\equiv importanza del job). → NB: è un fattore semplicistico.

→ FUNZIONE DI UTILITÀ

11/10/2022

Sistema Lava-metálico:

Il nostro scopo è definire un sistema su (che è un ambiente di esecuzione ma non un sistema operativo) real-time "BARE-METAL". Tale ambiente di esecuzione è minimale e auto-contenuto (\rightarrow durante la fase di bootstrap non possiamo assumere che esista nulla, né funzioni di libreria, né funzioni di sistema, perché appunto il SO non esiste). L'ambiente di esecuzione dovrà poi essere validato come HARD REAL-TIME.

L'hw su cui andremo a eseguire il nostro task è l'SBC (Single Board Computer), che si programma in modo simile ai nostri computer (ma ricordiamoci sempre dell'asincronia).

13/10/2022

Algoritmi per la Schedulazione real-time:

Sono divisi in 3 categorie:

- 1) CLOCK-DRIVEN
- 2) WEIGHTED ROUND-ROBIN
- 3) PRIORITY DRIVEN

Algoritmi clock-driven:

Sono algoritmi in cui le decisioni riguardo i job da eseguire e gli intervalli di tempo in cui questi devono terminare la esecuzione sono determinate in anticipo.

Qui in particolare:

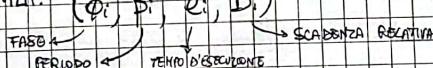
- Gli istanti in cui lo scheduler interviene sono fissati una volta per tutte.
- I task periodici sono conosciuti e costanti.
- Una schedulazione ottimale per i task può essere calcolata a priori.

Questo tipo di algoritmi è facile da implementare e verificare ed è efficiente. Tuttavia, un grosso svantaggio sta nel fatto che è poco flessibile (e.g. creare nuovi task a run-time è estremamente costoso).

Ciononostante, questi algoritmi permettono di trovare la soluzione migliore una volta per tutte (SCHEDULAZIONE STATICA). Se abbiamo task periodici, parliamo di SCHEDULAZIONE CICLICA.

Per quanto riguarda la schedulazione ciclica, il numero n di task nel sistema è fissato, e i parametri di tutti i task periodici sono conosciuti a priori.

ESPRIMIAMO CON UNA QUADRUPLA: (ϕ_i, p_i, e_i, D_i)



descrivendo
completamente
il task

Se usiamo una ferma, viene omessa la fase (implicitamente 0).
Se usiamo una coppia, viene omessa anche la scadenza relativa (implicitamente pari al periodo).

Dato un insieme di task, come facciamo a derivarne una schedulazione fattibile?

- 1) Calcolare l'iperperiodo.
- 2) Trovare una schedulazione fattibile in un iperperiodo, in modo tale da ripeterla all'infinito.

Questo è uno scheduler a tabella: infatti, viene usata una tabella che associa i tempi con i job/task da eseguire in quei tempi.

Ciononostante, in generale è preferibile lavorare con scheduler che soddisfino certe proprietà.

- Affiancare dello scheduler a intervalli regolari.

→ Distribuzione regolare degli intervalli IDE in un per periodo.

Queste sono proprie struttura, soddisfatte dagli scheduler detti CYCLIC EXECUTE (= cicli strutturati).

SCHEDULER CYCLIC EXECUTE:

Qui il tempo è partitionato in FRAME, all'interno dei quali è definita una lista di job da eseguire in sequenza (scacco di SCHEDULAZIONE). All'interno dei frame non si possono interrompere i frame job.

→ Noi vogliamo che l'iperperiodo sia multiplo di un frame. Per ottenere ciò, la fase di ogni task periodico deve essere ~~un~~ multiplo del frame.

→ Noi vogliamo anche che la durata di ogni job cada in un frame diverso della scheduler: da una parte, tutti i job devono cadere all'interno di un unico frame e, dall'altra, lo scheduler controlla che i job abbiano finito solo all'inizio di ogni frame.

~~~~~ MORALE DELLA FAVOLA: trovare la lunghezza del frame adeguato non è banale.  
Vedere le slide per le condizioni necessarie.

→ Se ho periodi con MCD troppo grande, si provi ad abbassare opportunamente il periodo di alcuni job (se possibile).

→ Se ho periodi ~~non~~ non interi, basta cambiare unità di misura del tempo per non avere tali periodi interi.

#### PRINCIPIO CHE VALE SEMPRE:

Minimizzare i tempi di risposta dei job periodici ~~in soft-real-time~~ soft-real-time è un obiettivo da conseguire.

Una tecnica che si usa è lo SLACK STEALING. → Vedete bene le slide.

È comunque una tecnica che non viene applicata per i priority scheduling.  
NB: I job periodici devono essere interrumpibili. Se non lo sono, non si possono gestire.

Per quanto invece riguarda i job aperiodici hard-real-time, devono rispettare assolutamente le scadenze. Per questo motivo si deve introdurre la possibilità di rifiutare la

YCLIC schedulazione di un job aperiodico e priori nel caso in cui non ci sia modo che rispetti le scadenze.

→ Ad esempio possono essere implementati dei test di accettazione.

→ Un modo efficiente per schedulare i job aperiodici hard real-time è ordinare in base alla loro SCADENZA. In particolare, si hanno due code:

- 1) Job rilesinati e non ancora accettati.
- 2) Job accettati.

L'algoritmo che si basa sulla scadenza + vicina è detto EDF.

Job per allineamento di 4 byte:  
sistema in scan

$= ALIGN(4);$

20/10/2022

### ALGORITMI PRIORITY-DRIVEN

Algoritmo Round-Robin:

Provedono la gestione dei job tramite delle code FIFO. Ogni job viene eseguito per un quanto di tempo (TIME-SLICE) ben definito; dopo di che il job viene deschedulato e si passa al job successivo, e così via circolarmente (VEDI SISTEMI OPERATIVI DELLA TRIENNALE).

Un algoritmo Round-Robin è detto PESATO (WEIGHTED) se a job differenti possono essere assegnati dei pesi diversi che determinano la lunghezza del relativo time-slice.

VANTAGGI: Sono fair, prevedono la starvation e sono molto semplici ed efficienti.

Svantaggi: Non considerano eventuali scadute dei job (per cui gli algoritmi di scheduling real-time hanno a picco); inoltre, non gestiscono bene i job con vincoli di precedenza.

Ma cos'è un algoritmo priority-driven?

→ È un algoritmo che non lascia mai intenzionalmente inutilizzato un processore (perciò è detto anche algoritmo WORK CONSERVING).

→ È un algoritmo in cui le decisioni dello scheduler vengono prese all'occasione di un evento ( $\rightarrow$  algoritmo EVENT-DRIVEN).



→ È un algoritmo con lo scheduler GREEDY.

Poi studieremo alcuni algoritmi priority-driven usando:

- Modello a task periodici (o sporadici).
- Numero di task prefissato.
- Singolo processore.
- Task indipendenti: non si hanno vincoli di precedenza o risorse condivise.
- Nessun job periodico.

PIÙ AVANTI NEL CORSO TORNEREMO AVERE VINCOLI DI QUESTI VINCOLI.

Gli algoritmi priority-driven ~~possono~~ essere realizzati seguendo dinamicamente valori numerici detti PRIORITÀ ai job. → Si può avere una lista di job ordinata in base alla priorità (per cui parliamo di LIST SCHEDULING).

Comunque sia, molti scheduler NON real-time sono priority driven, come ad esempio:

→ FIFO → QUI LA PRIORITÀ DIPENDE DALL'ISTANTE DI RILASCIO

→ LIFO

→ SJF (Shortest Execution Time First)

→ LRTF (Longest Execution Time First)

→ ROUND-ROBIN → QUI LA PRIORITÀ CAMBIA DINAMICAMENTE NEL TEMPO

Gli algoritmi priority-driven possono essere:

- di priorità fissa (FIXED-PRIORITY).
- di priorità dinamica (DYNAMIC-PRIORITY).

→ DINAMICO A LIVELLO DI TASK E STATICO A LIVELLO DI JOB

→ DINAMICO ANCHE A LIVELLO DI JOB → e.g. Round-Robin

#### TIPI FONDAMENTALI DI ALGORITMI PRIORITY-DRIVEN:

→ EDF: priorità inversamente proporzionale alla scadenza assoluta. → DINAMICO SOLO A LIVELLO DI TASK

→ LST: " " " allo slack del job. → DINAMICO A LIVELLO DI JOB

→ RM: " " " al periodo. → STATICO A LIVELLO DI TASK

→ DM: " " " alla scadenza relativa. → STATICO A LIVELLO DI TASK

→ LRT: Latest Release Time (= EDF inverso) è un algoritmo ~~NON~~ priority-driven.

#### Teorema:

Con un solo processore, job interrumpibili e nessuna condivisione sulle risorse condivise, gli algoritmi EDF, LRT, LST producono una schedulazione fattibile se è possibile far rispettare le scadenze di singoli job. In tal senso sono detti algoritmi OTTIMALI.

→ POTENZIALITÀ DI LRT: è più prevedibile perché tende a far completare tutti i job nei pressi della loro scadenza.

↳ Il problema è che non è ovunque: per ottenerlo, è necessario conoscere tutti i releases / scadenze dei job a priori.

Però

→ FRAZIONA DI LST: a differenza di LRT, EDF, richiede la conoscenza ANCHE DEL TEMPO DI ESECUZIONE dei job, il che complica le cose.

↳ LST ha due varianti: NON STRICT LST (dove le priorità dei job cambiano solo a volte di release / conclusione dei job) e STRICT LST (dove le priorità dei job sono modificate "continuamente").

Per questo perché + complesso e + costoso. \*NB: strict LST è difficile, non strict LST no.

→ Nel caso in cui le scadenze relative sono TUTTE implicate (coincidono col periodo), l'algoritmo RM si comporta in modo identico a DM.

↳ Non è l'unico caso: i due algoritmi coincidono anche se le scadenze relative sono in qualche modo "proporzionali" ai periodi.

→ In generale, DM è migliore di RM. Infatti:

- Se la schedulazione di DM è non fattibile, anche la schedulazione di RM è non fattibile.
- Esistono casi in cui la schedulazione DM è fattibile e la schedulazione RM no.

In ogni caso, né RM né DM sono ottimali.

↳ Però sono i più semplici da studiare e da implementare.

↳ FORSÉ EDF HA IL TRANS-OFF MIGLIORE TRA SEMPLICITÀ E OTTIMALITÀ.)

CIONONOSTANTE, RM E DM SONO TUTTORA UTILIZZATI NELL'INDUSTRIA.

Gli algoritmi priority-driven NON sono ottimali se i job sono NON interrumpibili.

27/10/2022

### Problema della validazione:

Dato un insieme di job, di processori e di risorse, la validazione mira a dimostrare l'esistenza di uno scheduling che rispetti tutte le scadenze dei job (= fattibile).

↳ Per gli algoritmi priority-driven il problema della validazione è difficile da risolvere a causa delle ANOMALIE DI SCHEDULAZIONE, che sono dei comportamenti temporali inattesi. Come mai? Perché con le anomalie diretta fortunata troverà il caso peggiore, e le dimostrazioni formali si basano appunto sul caso peggiore.

→ ADDITIVITÀ IL CASO PEGGIOR FOTREBBE NON ESSERE FACCIO

### ESEMPI DI ANOMALIE (NEL CASO DI JOBS NON INTERROMPIBILI):

- 1) Diminuzione della frequenza di rilascio di job.
- 2) Diminuzione del tempo di esecuzione di un job.
- 3) Aumento della velocità del processore.

### Esecuzione prevedibile:

Fissato un algoritmo, la schedulazione prodotta considerando i tempi di esecuzione massimi (minimi) per tutti i job è detta schedulazione massima (minima).

↳ L'esecuzione di un job è PREVEDIBILE se è sempre entro i limiti temporali stabiliti dalla schedulazione minima e massima.

### Teorema:

→ VUOLE ANCHE DIRE CHE NON RISORSE CONDIVISE

Un insieme di job INTERROMPIBILI, INDIPENDENTI CON ISTANTI DI RILASCO FISSATI schedulabili.

Lato su un processore da un algoritmo priority-driven è PREVEDIBILE.

Con + processori cosa cosa

↳ Sicuramente sarà l'indipendenza tra i job.

Non c'è alcuna anomalia; il caso peggiore esiste sempre.

Ci serve un modo per valutare le prestazioni degli algoritmi di scheduling...

### Fattore di utilizzazione:

Un algoritmo X ha un fattore di utilizzazione  $U(X) \in [0, 1]$  t.c. l'algoritmo può determinare una schedulazione fattibile per un insieme di task periodici su un processore se l'utilizzazione totale dei task è  $\leq U(X)$ .

Un algoritmo è tanto migliore quanto il suo fattore di utilizzazione è maggiore.

→ FIFO ha un fattore di utilizzazione dell'algoritmo pari a 0.

→ EDF ha un fattore di utilizzazione pari a 1. → Sarà la dimostrazione...

↳ Ma a questo punto perché dovremmo studiare altri algoritmi? Perché EDF non è proprio il più prevedibile.

#### Teorema:

Un sistema  $T$  di task indipendenti e interrumpibili con scadenze relative uguali ai rispettivi periodi e fattore di utilizzazione  $U_T$  ha una schedulazione fattibile su un singolo processore.  
 $\Leftrightarrow U_T \leq 1$ .

#### Corollario:

EDF ha fattore di utilizzazione  $U_{EDF} = 1$  per sistemi di task indipendenti, interrumpibili e con scadenze relative  $\geq$  rispettivi periodi.

#### Definizione:

La densità di un task  $(d, p, e, D)$  è il rapporto  $\frac{e}{\min(d, p)}$ .

#### Teorema:

Un sistema  $T$  di task indipendenti e interrumpibili e densità  $\Delta_T$  ha una schedulazione fattibile su un singolo processore se  $\Delta_T \leq 1$ .  
L'è una condizione sufficiente ma non necessaria per avere la schedulazione fattibile.

N.B.:  $U_T \leq \Delta_T$  SEMPRE.

#### Teorema:

Un sistema  $T$  con task periodici indipendenti, interrumpibili, con  $U_T < 1$  è schedulabile con EDF  $\Leftrightarrow \forall L > 0 \sum_{i=1}^n \left\lfloor \frac{L + p_i - d_i}{p_i} \right\rfloor \cdot e_i \leq L$ .

Gli algoritmi a priorità fissa sono peggiori di quelli a priorità dinamica risp. capacità di determinare schedulazioni fattibili. Ciò nonostante, esistono classi di sistemi che ammettono un algoritmo a priorità fissa ottimale.

ESEMPIO: sistemi di task semplicemente periodici (≡ ARMONICI).

→ Hanno periodi che sono uno il multiplo dell'altro.

Teorema:

Un sistema  $T$  di task ARMONICI, INTERRUPIBILI e INDEPENDENTI le cui scadenze relative  $\geq$  risp. periodi ha una schedulazione RM fattibile su un solo processore  $\Leftrightarrow \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$ .

Teorema:

Se per un sistema di task periodici, indipendenti e interrumpibili che sono in fase e hanno scadenze relative  $\leq$  rispettivi periodi  $\exists$  algoritmo a priori fissa che produce una sched. Bisogna anche la versione fattibile  $\Rightarrow$  anche DM produrrà una schedulazione fattibile.

02/11/10m00

void panic1 (void) {  
 panic(5);  
}

→ Disegna questa

→ RESET

→ UNDEFINED IN

08/11/2022

### Schedulabilità per RM e DM:

PROBLEMA: fissato un insieme di task e un algoritmo di schedulazione a priorità fissa (e.g. RM), come rischiare se l'algoritmo determinerà SEMPRE una schedulazione fattibile?

#### ISTANTE CRITICO:

Supponiamo che i job completino sempre entro l'istante di rilascio del job successivo del medesimo task ( $\leq$  entro il periodo). L'ISTANTE CRITICO del task è l'istante di tempo in cui il rilascio di un job  $t_k$  supera il massimo tempo di risposta possibile per quel job (è praticamente l'istante di tempo peggiore per il rilascio del job).

↳ Se almeno un job da  $T_i$  non rispetta la scadenza relativa, l'ISTANTE CRITICO è un momento in cui il rilascio di un job provoca il mancato rispetto della scadenza di quel job.

#### Teorema:

In un sistema con task a priorità FISSA e TEMPI DI RISPOSTA piccoli, l'istante in cui uno dei job di  $T_i$  viene rilasciato contemporaneamente ai job di tutti i task con priorità maggiore di  $T_i$  è un ISTANTE CRITICO.

↳ QUESTO TEOREMA È IMPORTANTE perché ci STA LANCIO IL CASO PEGGIORTE.

#### FUNZIONE DI TEMPO NECESSARIO:

Siano dati i task  $T_1, \dots, T_i$  in fase al tempo  $t_0$  con priorità decrescenti. Il tempo necessario per eseguire tutti i job dei task  $T_1, \dots, T_i$  nell'intervallo  $[t_0, t_0 + t]$  ( $t \leq p_i$ ) è:

$$W_i(t) = e_i + \sum_{k=1}^i \left\lceil \frac{t}{p_k} \right\rceil e_k$$

TEMPO DI ESECUZIONE DEI JOBS DEL TASK  $T_k$   
# RILASCI DI JOBS DEL TASK  $T_k$  NELL'INTERVALLO AMPIO  $t$ .

#### TEST DI SCHEDULABILITÀ:

Siano dati i task  $T_1, \dots, T_i$  di prima (che siano effettivamente schedulabili). Il task  $T_i$  può essere schedulato nell'intervallo di tempo  $[t_0, t_0 + D_i]$  se  $\exists t \in [t_0, t_0 + D_i] | W_i(t) \leq t$ .

↳ SE IL TEST FALLISCE, I JOBS DA  $T_i$  FOTREBBERO MANCARE LA PROPRIA SCADENZA (ma non necessariamente).

Diciamo che la scadenza non viene rispettata nel caso peggiore.

MASSIMO TEMPO DI RISPOSTA DI UN TASK:

È dato da:  $W_i = \min \{ t \leq D_i \mid t = w_i(t) \}$ .

Ma perché ci appropriano al test di schedulabilità anziché effettuare una simulazione sul caso peggiore?

→ Noi sistemi + complessi risulta + fastidioso mettere in piedi una simulazione.

Per giunta, analizzando un sistema di task con tempi di risposta non necessariamente piccoli è più difficile. Di fatto:

→ Ci possono essere nello stesso istante + job di uno stesso task in attesa di essere eseguiti.

→ Nei vari istanti critici non è più vero: perdiamo il caso peggiore.

BUSY INTERVAL:

Un INTERVALLO TOTALMENTE OCCUPATO di livello  $T_i$  è un intervallo di tempo  $(t_0, t_1]$  t.c.:

→ In tutti i job del task  $T_i$  rilasciati prima di  $t_0$  sono stati completati.

→ In un job di  $T_i$  è rilasciato.

→  $t_1$  è il primo istante in cui tutti i job di  $T_i$  rilasciati a partire da  $t_0$  sono stati completati.

→ NELL'INTERVALLO TOTALMENTE OCCUPATO IL PROCESSORE NON PUÒ MAI ESSERE IDE.  
ALTRIMENTI L'INTERVALLO TERMINE PERDE PRIMA.

TEST DI SCHEDULABILITÀ GENERALE:

Idea: se tutti i job di  $T_i$  si analizzano tutti i job di  $T_i$  eseguiti nel 1° intervallo totalmente occupato di livello  $T_i$ .

→ INIZIO DELL'INTERVALLO: istante di rilascio dei primi job dei task  $T_1, \dots, T_i$ .

→ LUNGHEZZA DELL'INTERVALLO: Missing

LEMMA:

Il tempo di risposta massimo  $W_{ij}$  del  $j$ -esimo job di  $T_i$  in un intervallo totalmente occupato di livello  $T_i$  in fase è uguale al minimo valore di  $t$  che soddisfa l'eqn:

$$t = W_{ij}(t + (c_{j-1})p_i) - (c_{j-1})p_i \quad \text{con} \quad W_{ij}(t) = j p_i + \sum_{k=1}^i \lceil \frac{t}{p_k} \rceil \cdot e_k$$

Condizioni di schedulabilità:

Potrebbe essere comodo stabilire se un insieme di task è schedutabile anche se periodi di

esecuzione / scadenze non sono noti.

Condizione di Liu-Layland:

Un sistema di  $n$  task indip., interrappresentabili, con  $D_i = p_i$  può essere schedulato secondo l'algoritmo RM se il ~~se~~ fattore di utilizzo  $U_T$  del processore è minore o uguale a

$$U_{RH}(n) = m \left( 2^{1/n} - 1 \right)$$

$$\Rightarrow \lim_{n \rightarrow \infty} U_{RH}(n) = \ln 2 = 0,693$$

Se la condizione non è vera, NON È DETTO CHE L'INSIEME DI TASK NON SIA SCHEDULABILE.

Test periodico:

È un test migliore: un sistema di  $n$  task (con le stesse caratteristiche di cui sopra) può essere schedulato con RM se:  $\prod_{k=1}^n \left( 1 + \frac{e_k}{p_k} \right) \leq 2$

Condizione di Knut-Munk:

Un sistema di ~~n~~ task periodici, indip., interrappresentabili, con  $p_i = D_i$  può essere partizionato in  $M$  sottoinsiemi disgiunti  $Z_1, \dots, Z_M$ , ciascuno dei quali contiene task semp. periodici allora il sistema è schedulabile con RM se: MISSING

$$w = t \\ buf[-i] = (char)(t + '0');$$

?

$$w += puts(buf + i);$$

return w;

?

Consideriamo ora tutti gli interi (anche negativi):

int putd (long v) {

int w=0

if (v<0){

$$w += putc(' - ')$$

$$v = -v;$$

?

$$w += putv(v);$$

return w;

?

Numeri in modo mobile:

f

15/11/2022

### Schedulazione di job bloccanti e aperiodici:

Il rallentam./ritardo nell'esecuzione di job può essere di due tipi:

1) TEMPO DI BLOCCO  $\equiv$  tempo in cui il job, pur essendo stato rilasciato, non può essere eseguito per qualche motivo esterno.

↳ Il TEMPO MASSIMO DI BLOCCO  $b_i$  è la lunghezza massima dell'intervallo in cui un job di  $T_i$  può essere bloccato.

dAllora, a tra  
da:

Ma attenzione  
job si auto-

se job sa  
→ Utilizza  
→ Interag

2) RALLENTAMENTO SISTEMATICO: può essere ad esempio il tempo richiesto per eseguire lo scheduler o per effettuare un context switch.

#### AUTO-SOSPENSIONE:

È data da operazioni bloccanti che "auto-bloccano" il job (e.g. accesso al disco rigido).  
↳ Un caso facile per battere l'auto-sospensione è quello in cui avviene l'appena il job viene rilasciato. Qui è sufficiente impostare l'istante di rilascio pari a  $p_i + x$  e la scadenza relativa pari a  $D_i - x$ .

NB: L'auto-sospensione fa l'idea di essere occulto nei confronti degli altri job tramite il  $\Delta t$  lascio della CPU. Invece, anche se è contro-intuitivo, è un danno (in particolare può provocare danni ai task a cui priorità inferiore).

NB: L'auto-sospensione (anche nel caso facile) non si può trarre tramite i task specifici: ricordiamo che gli sporadici prevedono un certo intervallo di tempo minimo tra un rilascio e l'altro (cosa che non avviene con l'auto-sospensione).

Ora ci poniamo questa domanda: qual è il danno (il ritardo) massimo che può provocare l'auto-sospensione?

CASO 1: Tempo auto-sospensione > durata del job  $\Rightarrow$  RITARDO MAX = durata del job.

CASO 2: Tempo auto-sospensione < durata del job  $\Rightarrow$  RITARDO MAX = tempo auto-sospensione.

Dato un task  $T_k$ , sia  $x_k$  il tempo massimo di auto-sospensione di ogni job di  $T_k$ .

Allora, il rallentamento inflitto a un job  $T_i$  (con priorità peggiore) da parte di  $T_k$  è dato da:

$$b_i(ss) = x_k + \sum_{t=1}^{i-1} \min(p_t, x_k)$$

Ma affermare: è importante conoscere anche il numero massimo di volte in cui ciascun job si auto-sospende.

#### NON INTERRUZIBILITÀ DEI JOBS:

I job sono non interruzibili ad esempio quando:

→ Utilizzano una risorsa critica condivisa.

→ Interagiscono con dispositivi hardware.

E così via

### DEFINIZIONE:

Un job  $J_i$  è bloccato per INTERROPIBILITÀ quando è pronto all'esecuzione ma non può più essere eseguito a causa di un job con priorità peggior non interrompibile.

↪ Nell'intervallo di tempo in cui avviene questo si ha un'inversione di priorità.

→ La NON INTERROPIBILITÀ fa danni agli altri job, in particolar modo con scheduler work-conserving.

Ma quant'è la durata il ritardo max causato da un job non interrompibile? (sicuram.  $\leq$  della sua durata).

$$\hookrightarrow b_i(np) = \max \{ \Delta_k \text{ per ogni task } T_k \text{ di priorità minore di } T_i \}$$

dove  $\Delta_k = \text{tempo d'esecuzione massimo della + lunga scorruta non interrompibile del job di } T_k$ .

↪ Di fatto, ciascun job può essere bloccato per non-interrompibilità da al + un altro job di priorità peggior; poi il controllo va necessariamente a lui.

↪ È PER QUESTO CHE ABBIANO IL MAX E NON LA SOMMA

→ Se indichiamo con  $b_i$  il ritardo massimo totale che si può sperimentare, abbiamo:

$$b_i = b_i(ss) + (K_i + 1) \cdot b_i(np)$$

Qui il # max di auto-sospensioni è già incluso.

# MAX AUTO-SOSPENSIONI  
( $i+1$  è il rilascio iniziale)

→ Le si possono avere anche alla terminazione  
dei job non interrompibili.

→ Indichiamo con  $CS$  il costo di un cambio di contesto.

LST è particolarmente inefficiente se  $CS$  è grande.

Davanza: come estendiamo il test di schedulabilità per trattare i job che possono bloccarsi?

↪ L'idea è trattare i tempi di blocco come tempo di esecuzione in più.

In pratica alla funzione di tempo richiesto  $W(f)$  va aggiunto un addendo  $b_i$ . Analogamente per il test di schedulabilità generale.

↪ ATTENZIONE: qui ci sommiamo sempre  $b_i$  e non  $j \cdot b_i$  perché  $b_i$  NON è un vero e proprio tempo di esecuzione.

### CONDIZIONI DI SCHEDULABILITÀ PER TASK BLOCCANTI A PRIORITY FISSA:

Perche' ciascun job può bloccare in misura differente, bisogna applicare la condizione di schedulabilità task per task.

### Teorema di Baker:

In una schedulazione EDF, un job con scadenza relativa  $D$  può bloccare un altro job con scadenza relativa  $D'$  solo se  $D > D'$ .

### Teorema di Baker con auto-sospensione:

In una schedulazione EDF, un job con scadenza relativa  $D$  può bloccare un altro job con scadenza relativa  $D'$  mediante un'auto-sospensione di durata  $\tau_1$  solo se  $D > D' - e_1 - \tau_1$ .

### Schedulazione basata su tick:

TICK = interattivazione periodica di uno scheduler time-driven.

Qui si hanno due tipi di job: quelli PENDENTI (non ancora riconosciuti dallo scheduler) e quelli ESECUTAVILI.  $\rightarrow$  Un job pendente diventa eseguibile all'arrivo di un tick.

$\hookrightarrow$  Qui il tempo in cui bisogna aspettare il tick per diventare un job eseguibile può essere anche molto lungo come un blocco.

In realtà, quello che facciamo è:

1) Aggiungere un task  $T_0$  di priorità massima.  $\rightarrow T_0 = (p_0, e_0)$

2) Aggiungere un task  $T_{0,k} = (p_k, CS_0)$  con priorità minore di  $T_0 \quad \forall k = i+1, \dots, n$

3) Aggiungere  $(K+1)CS_0$  al tempo di esecuzione di  $T_k \quad \forall K = 1, \dots, n$ .

4) Fissare  $b_i(h_p) = \left( \frac{T_{i+1} - T_i}{T_{i+1} - T_i + \tau_{i+1}} \right) \cdot p_0$ .

$\tau_{i+1}$  = tempo iniziale

### Schedulazione priority-driven di job aperiodici:

Ricordiamo che i job aperiodici possono essere hard-real-time o soft-real-time. In base alla categoria, richiedono algoritmi differenti.

### JOBS APERIODICI SOFT-REAL-TIME:

Il modo più semplice per eseguirli è in BACKGROUND ( $\rightarrow$  esecuzione dei job aperiodici

solo quando il processore è idle).

↳ È CORRETTO MA NON OTTIMALE.

Soddisfa tutte le rispettive

potrebbe esserci modo di anticipare i job aperiodici senza che quelli periodici manchino le scadenze.

Un algoritmo corretto e ottimale è la SCHEDULAZIONE CON SLACK STEALING: esegue gli job aperiodici finché lo slack globale è  $\geq 0$ . Il problema è che è un algoritmo complesso e non utilizzato in pratica.

Ottieniamo poi la schedulaz. basata su polling, la cui correttezza dipende dai parametri del poller ma sicuramente non è ottimale.

↳ Qui il servizio di polling controlla periodicamente la coda dei job aperiodici. Se è vuota, va in dormire. Se nel frattempo arriva un job aperiodico, deve aspettare che il servizio si risvegli.

iomemdef(DTTHERO - IRQS  
" " "  
IRQEN  
IRQE  
TCLR  
TLDR  
TTGR

#define TCAR\_IT\_FLAG (

#define QVF\_IT\_FLAG (

#define MAT\_IT\_FLAG

All'interno di un altro

void init\_ticks (void

irq\_disable();

if (register\_isr

29/11/2022

RIPRENDIAMO IL DISCORSO SULLA SCHEDULAZIONE DI JOBS PERIODICI CON POLLING..

#### SERVER PERIODICI:

Sono una classe di task periodici avvertiti.

→ Periodo  $p_s$ , budget  $e_s$ , dimensione  $U_s = e_s/p_s$ .

→ Regola di consumo (come il budget viene consumato).

→ Regola di rifornimento (come il budget viene ripristinato).

Il polling è assimilabile a un server periodico (vedi slide).

→ Il problema del server di polling è che il budget è perso quando la coda di job (aperiodica) diventa vuota.

→ Gli algoritmi che entrambi questo problema sono detti a CONSERVAZIONE DI BANDA.

#### Server precastimabile:

È il + semplice algoritmo a conservazione di banda.

→ L'inizio del success. periodo

↳ Viene fuori che il server f

$e_s + f$

per il calcolo di  $W_i(t)$  (if nella formula.

→ Di fatto, il caso peggiore priorità massima.

→ Esistono apposta per il

→ Esistono apposta per il

#### Server sporadico:

È un altro algoritmo a di quelli precastimabili,

REGOLA DI CONSUMO: il budget è decrementato per il tempo di esecuzione del job. Quindi il budget si conserva, ma si attesta come invece avviene con il pitter.

REGOLA DI DISTRIBUIMENTO: il budget è impostato a  $e_s$  ( $\approx$  BUDGET MASSIMO) agli istanti

$$K \cdot p_s, K = 1, 2, \dots$$

Significa che il budget fissa al valore massimo  $e_s$  a ogni periodo: in ciascun periodo è possibile mantenere il cronometro task aperiodico per al più  $e_s$  unità di tempo.

È possibile applicare il test di schedulabilità con i task a server precastinatibile MA il server precastinatibile non è proprio la stessa cosa rispetto ai task periodici.

Lemme: → si tratta di trovare il caso peggiore

In un sistema di task periodici, INDIF, INTERCOMP., con priorità fissa  $D_i \leq p_i$ , e con server precastinatibile  $(p_s, e_s)$  con priorità MAX, un istante critico di un task  $T_i$  si verifica all'istante  $t_0$  se:

→ A  $t_0$  è rilasciato un job di tutti i task.

→ A  $t_0$  il budget del server è  $e_s$ .

→ A  $t_0$  è rilasciato almeno un job aperiodico che impegna il server da  $t_0$  in avanti.

→ L'inizio del successivo periodo del server è a  $t_0 + e_s$ .

Esiste fuori che il server precastinatibile risulta occupato per la seguente età di tempo:

$$e_s + \frac{t - t_0}{p_s} \cdot e_s$$

→ # periodi; prendo la parte intera superiore perché il task aperiodico ha priorità massima.

Per il calcolo di  $w_i(t)$  (il tempo necessario), basta solo aggiungere questo addendo nella formula.

→ Di fatto il caso peggiore è proprio dato dal fatto che i task aperiodici hanno priorità massima.

→ È tecnicamente opposta per il caso RTT con server precastinatibile.

→ È un altro tecnicamente opposta per il caso EDF con server precastinatibile.

Server sporadici:

È un altro algoritmo a conservazione di banche. I server sporadici, a differenza di quelli precastinatibili, sono perfettamente assimilabili ai task periodici, per-

cui non richiedono una complicazione del modello analitico. Però sono complicati assai da implementare. Da qui introduciamo una variante, ~~CBS~~ <sup>new algorithm</sup> ~~per il server~~ <sup>CBS è:</sup> ~~sporadico semplice.~~ <sup>La sua</sup> <sup>sia.</sup>

### ~~CBS~~ Server sporadico semplice:

→ REGOLA DI CONSUMO:  $\forall t > t_0$  (è l'istante ultimo rifornimento), il budget è decrementato di 1 unità di tempo se vale una delle seguenti condiz.:

c1) Il server è in esecuzione.

c2) Il server è stato in esecuzione dopo  $t_0$  e ora non sta eseguendo mai per colpa di task di priorità ~~minore~~ migliore (beni per altri motivi). \*

→ REGOLA DI RIFORNIMENTO: il prossimo rifornimento avviene a  $t_0 + p_s$ , con due eccezioni:

1) Se non c'è niente da eseguire, il budget è rifornito nell'istante iniziale dell'intervalle  $t_0, t_0 + p_s$ .

2) Se il job rimane in attesa per più unità di tempo di  $p_s$ , viene premiato.

Poi ci sono altri due regole da verificare sulle liste.

\* È questa caratteristica che rende il server sporadico semplice perfettamente assimilabile ai task teorici.

### Altra variante: server sporadico background:

di differenza della variante semplice, esegue sempre job aperiodici se nessun task periodico è eseguibile; quando il budget non decremente mai nel caso in cui nessun task periodico è eseguibile.

Questa variante NON conviene solo nel caso in cui si hanno tipologie di task aperiodici statiche differenti che richiedono servizi sporadici differenti ( $\rightarrow$  qui sceglieremo al più 1 server sporadico/~~background~~).

### CBS:

È l'ultimo algoritmo a conservare la banda che vediamo. Qui il server per i job aperiodici è integrabile in uno scheduler a priorità fissa a livello di job. È basato sulla priorità dinamica a livello di task; è work conserving.  $\rightarrow$  l'occupaz. del processore del server non supera mai la frazione di tempo media minita (bandwidth costante).

modello analitico. Però so complicati

cui dobbiamo definire anche la scadenza assoluta corrente  $d_s$ .

iamo una variante, ~~per il server~~, il SERV

↪ CBS è schedulato con EDF insieme ai task periodici considerando proprio  $d_s$ .

↪ La scadenza viene allungata (è si abbassa la priorità) sia se il budget si riconsumo, sia se il budget risulta troppo alto (i.e.  $C_s > (t_s - t) \cdot U_s$  in un istante di tempo  $t$ )

mo riconsumo), il budget è decrementato  
seguenti condiz.:

↪ È QUESTO CHE PERMETTE CHE IL SERVER STA ASSIMILABILE AL TASK PERIODICI.

Schedulabilità EDF di job aperiodica hard-real time:

Ricordiamo che qui abbiamo anche un'ACCETTABILITÀ di un job aperiodico ( $\rightarrow$  se e ora non sta eseguendo non per colpa si per altri motivi). \*

Teorema:

↪ avviene a tempo  $t_s$ , con due eccezioni  
Un sist. di job aperiodici indip. e interramp. è schedulabile con EDF se la densità  
nella scadenza dell'intervallo ritmico,  $\alpha_{ap}$  ( $\alpha = \frac{\text{TEMPO ESECUZ.}}{\text{SCADENZA - ISTANTE RILASCO}}$ ) totale di tutti i job attivi (nella scadenza) è scadenza  
venero premiato.

Ad.

dico semplice perfettamente assi

↪ Grazie a questo teorema possiamo costruire un test di accettazione, tramite il quale si va proprio a guardare la densità totale dei job nei vari intervalli di tempo. Se c'è almeno un intervallo in cui la densità è  $> 1$  NON è detto che i job non siano schedulabili!

↪ Lo condiz. del teorema è solo sufficiente.



28/11/2022

### Controllo d'accesso alle risorse condivise:

Le risorse condivise possono introdurre ritardi finora non considerati. Consideriamo risorse riciclabili seriali di tipo  $R_1, \dots, R_p$ , dove ogni  $R_i$  ha  $n_i$  unità di risorsa indistinguibili.

→ Ciascuna unità di risorsa può essere occupata al più da un job.

→ I job possono anche prendere + unità di risorsa contemporaneamente.

→ Richiesta per accedere a una risorsa  $R_i$  da parte di un job:  $L(R_i, n)$ , dove  $n = \#$  UNITÀ DI RISORSA RICHIESTE.

↳ Se la richiesta è soddisfatta  $\Rightarrow OK$

affermamento  $\Rightarrow$  il job viene bloccato, per cui sperimenta del ritardo.

→ Rilascio di una risorsa  $R_i$ :  $U(R_i, n)$

→  $L(R_i, n) = L(R_i)$ ;  $U(R_i, n) = U(R_i)$

Si dice che due job hanno un conflitto di risorse se entrambi richiedono una risorsa dello stesso tipo.

Due job, invece, si contendono UNA risorsa se entrambi richiedono una risorsa dello stesso tipo CONTEMPORANAMENTE.

→ Sezione critica: va dalla richiesta di risorse al loro rilascio.

Le richieste di risorse da parte di uno stesso job possono essere cumulate, ma c'è un limite per cui il rilascio delle risorse stesse avviene secondo una logica LIFO.

→  $[R_1, n_1; e_1 [R_2, n_2; e_2]]$  è una notazione che sta a indicare che c'è una sezione critica esterna di lunghezza  $e_1$  e una sezione critica interna di lunghezza  $e_2$ .

Con le risorse si può avere un'inversione di priorità nel momento in cui il job di un task con priorità migliore richiede una risorsa precedentemente acquisita dal job di un task con priorità peggiore.

- NB: Le risorse sono anche oggetto di ANARCHIA DI SCHEDULAZIONE.
- NB2: Le risorse, oltre alle semplici inversioni di priorità, ~~o~~ possono anche alle INVERSIONI DI PRIORITÀ NON CONTROLLATE ( $\rightarrow$  arbitrariamente lunghe: di fatto, i job dei task di priorità MAGGIORI, a causa dell'attesa per la risorsa, possono subire starvation).
- NB3: Quando si ha a che fare con più di una risorsa, si può andare anche incontro ai deadlock.
- Per via di NB2 e NB3, bisogna prendere delle accortezze particolari per la schedulazione dei job.  
↳ Comunque sia, ci preme di più l'inversione di priorità non controllata.
- Protocollo NPCS:
- In un job acquisita una risorsa assegnata non può essere interrotto.  
 $\hookrightarrow$  FUNZIONA ma è possibile che NON ci siano AUTOSOSPENSIONI nella sezione critica.  
Nel senso che previousi sia inversioni di priorità non controllate sia i deadlock.
- Quando andiamo ad analizzare i task, tra i vari tempi di blocco, c) ritorniamo ad ~~essere aggiungere~~ il TEMPO DI BLOCCO PER CONFLITTO DI RISORSE, che indichiamo con  $b_i$  (rc).
- LIMITE DI NPCS:  
Se è ottimale: il job con risorsa blanda fa priori i job di priorità ~~opp~~ migliore a cui non frega niente della risorsa.
- PO. Protocollo priority-inheritance:  
Evita l'inversione della priorità non controllata ma non il deadlock.  
IDEA: alzare la priorità del job con la risorsa\* in modo tale che i job dei task con priorità migliore NON subiscano starvation o causino della schedulazione dei job di priorità intermedia nel frattempo.  
\* QUESTO AVVIENE SOLO NEL MOMENTO IN CUI C'È UNA CONTESSA DELLA RISORSA.
- job LIMITI DI PRIORITY-INHERITANCE:  
- Come dicevamo, non evita i deadlock.

- Non ridurre i tempi di blocco dovuti ai conflitti sulle risorse al minimo teoricamente possibile.

#### Protocollo priority-ceiling:

È adatto solo agli scheduler con priorità fissa. Evita sia l'interazione di priorità non controllata, sia il deadlock.

IDEA: associare a ogni risorsa  $R$  il valore **PRIORITY CEILING**  $\bar{\pi}(R)$  pari alla max priorità dei job che usano  $R$  (dico, tenendo le caratteristiche del protocollo priority-inheritance).

→ Il sistema mantiene anche il **CURRENT PRIORITY CEILING**  $\hat{\pi}(t)$ , che è la max priorità  $\bar{\pi}(R)$  fra tutte le risorse del sistema correntemente in uso al tempo  $t$ .

COME SERVE? → quando un job (con priorità  $\pi(t)$ ) richiede una risorsa, si ha  
dunque se è libera e se.

→  $\pi(t) > \hat{\pi}(t)$ .

→ Il job ha una risorsa con priority ceiling pari a  $\hat{\pi}(t)$ .

Ma come fa sì che l'algoritmo a meno che i deadlock? Impedendo un ordinamento UNICO delle risorse a che possono essere accedute ( $\rightarrow$  non è possibile che job<sub>1</sub> richieda prima  $R_1$  e poi  $R_2$  e, nel frattempo, job<sub>2</sub> richieda prima  $R_2$  e poi  $R_1$ ).

→ SE proprio questo deve succedere, priority ceiling evita che job possieda  $R_1$  mentre job possiede  $R_2$ .

01/12/2022

È giunto l'ora di implementare lo scheduler con job intercambiabili.

Iniziamo col definire uno stack per ogni task. Al cambio di contesto, sappiamo che i registratori  $R_0, R_1, R_2, R_3, R_4, R_5, R_6$  vengono salvati sullo stack di per sé; tutti gli altri registratori vanno memorizzati esplicitamente nel DESCRITTORE DEL TASK.

Inoltre, il cambio di stack implica il cambio dello stack pointer sp.

→ Per cambieremo il contesto alla terminazione della funzione `bsp_irq()` interna alla gestione dell'interrup.

→ ATTENZIONE: il cambio di contesto si può fare solo se ci troviamo nel' interruzione esterna, nel caso in cui si abbia un annidato di interruzioni.

→ nel caso No switch (.Lhoswitz) deve fare un po' di giochi di presto:  
carica Q nel registro  $r_0$ ; poi carica  $\sqrt{r_0}$  nel registro  $r_1$ , e.g.  $r_0 \leftarrow \text{valore stack}$ ; valore dei  
register incrementa il valore di sp in modo da puntare alla cima dello  
stack. Infine, carica in pc il valore di  $sp+8$  e sp torna a puntare al valo-  
re X. → VEDI STA CASA CON LE SLIDE SE NO MUORI.

06/12/2022  
Compleiamo il discorso sul controllo d'accesso alle risorse condivise e, in particolare, sul  
protocollo priority-ceiling...

- Esistono cause di blocco per l'accesso alle risorse:
  - blocco diretto: la risorsa è già occupata.
  - priority-inheritance: il job viene bloccato perché un job  $j_1$  con priorità maggiore di  $j_2$   
usa la risorsa e un job  $j_2$  con priorità teoricamente inferiore si ritrova con  
priorità innalzata.
  - priority-ceiling: il job viene bloccato quando richiede una risorsa e.g. libera una la sua priorità  
è inferiore al ceiling del sistema.

Teorema:  
Col protocollo priority-ceiling un job può essere bloccato al massimo per la durata  
di UNA sezione critica.

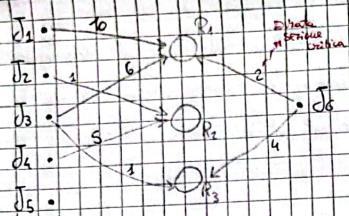
Se un job viene bloccato, è bloccato da un solo altro job.  
Non esistono blocchi transitivi ( $j_1$  blocca  $j_2$  che blocca  $j_3$  ...).

Da qui, il tempo di blocco per il conflitto di risorse  $b_i(cc)$  è il massimo ritardo  
di un job del task  $T_i$  causato da un conflitto di risorse.

Per il calcolo di  $b_i(cc)$ , basta determinare il max tra ritardo da blocco di  
ritardo, ritardo per priority-inheritance e ritardo per priority-ceiling.

Per farlo, campi delle tabelline.

vedere esempio pag. dietro =>



Blocco per INHERITANCE

|                | J <sub>2</sub> | J <sub>3</sub> | J <sub>4</sub> | J <sub>5</sub> | J <sub>6</sub> |
|----------------|----------------|----------------|----------------|----------------|----------------|
| J <sub>1</sub> |                |                |                |                |                |
| J <sub>2</sub> | *              | 6              |                |                | 2              |
| J <sub>3</sub> | *              | 5              |                |                | 2              |
| J <sub>4</sub> |                | *              | 4              |                |                |
| J <sub>5</sub> |                |                | *              | 4              |                |

Ho preso i 2 e 6 nella tabella del floor diretto e l'ho progettato giù

Ho preso il 2 e il 4 nella tabella del blocco diretto e l'ho progettato giù

NB: se abbiamo job con la stessa priorità, le cose si complicano: ragioniamoci su.

Problemi del priority-ceiling:

STACK-BASED PRIORITY CEILING:

È una semplificazione tale per cui tutti i job condividono il medesimo stack.

→ PROBLEMA: supponiamo che prima j<sub>1</sub> e poi j<sub>2</sub> richiedano una risorsa condivisa. j<sub>2</sub>, quando va in esecuzione, si rithatta la risorsa occupata, per cui dovrà restituire il controllo; Ma a questo punto, lo stack ormai è stato sparato da j<sub>1</sub> (dovendo che esso concludeva l'esecuzione "con successo"), per cui risulta inutile restituirlo da j<sub>2</sub>.

Per evitare questo problema, lo stack-based priority ceiling cambia la regola

Blocco DIRETTO

|                | J <sub>2</sub> | J <sub>3</sub> | J <sub>4</sub> | J <sub>5</sub> | J <sub>6</sub> |
|----------------|----------------|----------------|----------------|----------------|----------------|
| J <sub>1</sub> |                |                |                |                |                |
| J <sub>2</sub> | *              |                |                | 5              |                |
| J <sub>3</sub> |                | *              |                |                |                |
| J <sub>4</sub> |                |                | *              |                |                |
| J <sub>5</sub> |                |                |                | *              |                |

di sched

NB: con q  
non c'è m  
che divede

NB2: po

CEILING-  
E il pro  
Systems

Anche  
ceiling,  
-priorità

Teorema  
per sto

nel caso di job con tutte priorità div  
se, è falso farlo alla tabella del blocco  
inherisce eccetto per i job che non ab  
una alcuna risorsa.

) → NPC:

→ PRIO  
de

→ PRI

→ SI

→ CE

NPCs  
Tempo  
Lg

della schedulazione rispetto al priority-ceiling (VEDERE SUDE).

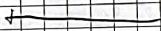
NB: con questo protocollo, se un job richiede una risorsa, questa è SEMPRE libera: non c'è mai contesa. Anzi, quando inizia la propria esecuzione, tutte le risorse che chiederà sono libere  $\Rightarrow$  il protocollo NON HA TEMPI DI ACCESCI DIRETTI.

NB2: per questo protocollo, è NECESSARIO che i job NON si autosospendano.

#### CEILING-PRIORITY:

È il protocollo utilizzato nella libreria per il accesso alle risorse condivise (Real-Time Systems Annex) del linguaggio di programmazione Ada 95.

Anche qui non c'è mai contesa, anzi: ceiling-priority e stack-based priority-ceiling, senza autosospensione, sono IDENTICI. Tuttavia, è possibile adattare ceiling-priority al caso in cui i job si autosospendano.



#### Teorema:

I tempi di blocco massimi  $b_i(\text{rc})$  dovuti ai conflitti di risorse per priority-ceiling e per stack-based priority-ceiling sono identici.

live

2

3

4

Ma quando possiamo usare l'autosospensione?

→ NPCS: NO se siamo all'interno di una sezione critica.

→ PRIORITY-INHERITANCE: sì ma a patto che il job che possiede la risorsa e si autosospende non subisca un abbassamento della priorità.

→ PRIORITY-CEILING: come priority-inheritance.

→ STACK-BASED PRIORITY-CEILING: NO NO NO.

5a

6a

7a

8a

9a

→ CEILING-PRIORITY: se un job si auto-sospende in una sezione critica, nessun job ha priorità minore o uguale può essere eseguito.

10a

11a

12a

13a

tempo di blocco da autosospensione =  $b_i = b_i(\text{ss}) + (k_i + 1) \cdot \max \{ b_j(\text{mp}), b_j(\text{rc}) \}$

Qui consideriamo il tempo di blocco per l'autosospensione equivalente al tempo di blocco dovuto alla conflitto delle risorse.

PRIORITY-CEILING E CEILING-PRIORITY:

$$\text{Tempo di blocco con autosospensione} = b_i = b_i(ss) + (K_i + 1)(C_i(N_p) + b_i(T_C))$$

Priority-ceiling in sistemi a priorità dinamica:

Qui c'è un problema: è calcolato il priority-ceiling delle risorse, che risulta essere non costante: il priority-ceiling del sistema cambia anche ogni volta che viene rilasciato un nuovo job. → È UN INCUBO CHE FUNZIONA.

Nella pratica, in caso di priorità dinamica (sia essa a livello di task o a livello di job), è molto più indicato ricorrere a protocolli come NPCS o PRIORITY-INHERITANCE.

Accesso alle risorse condivise da parte dei job aperiodici:

PROBLEMA: è possibile che il servizio che gestisce i job aperiodici esaurisca il budget mentre un job aperiodico sta utilizzando la risorsa condivisa.

→ Manca possono forse in modo che i job periodici vengano bloccati da quello aperiodico per un tempo indeterminato.

In realtà, la soluzione consiste nel far continuare l'esecuzione del job aperiodico fin tanto che la risorsa non viene rilasciata (anche se il budget è esaurito).

In tal caso, si considera che il budget vada in negativo, in modo tale che il servizio si penalizzi nei riguardi successivi.

MA → sul momento, il job aperiodico ha comunque fatto un danno.

Perciò, nel controllare la schedulabilità, si deve considerare tra i tempi di blocco anche il tempo d'esecuzione della più lunga sequenza critica dei job aperiodici.

13/12/2022

La funzione sys.schedule():

È inserita in modalità SYSTEM, e non da parte di un handler che si occupi di alcune registrazioni: dobbiamo salvare noi i registri sullo stack.

Analizziamo tale funzione:

c.v. quando sto per selezionare un job inviato un quanto una un appena eseguito.

bisognerebbe

15/12/2022

## SISTEMI REAL-TIME MULTIPROCESSORE

Sono sistemi con + processori, che possono essere dello stesso tipo ( $\rightarrow$  eseguono job dello stesso tipo) oppure di tipo diverso ( $\rightarrow$  eseguono job di tipo differente).

Nei esamineremo il caso in cui abbiamo n processori dello stesso tipo.

$\hookrightarrow$  che comunque è una complicazione pesante rispetto al caso del singolo processore.

Vantaggi:

$\rightarrow$  favoriscono

$\rightarrow$  flusso

Definizione:

SISTEMA REAL-TIME STATICO = sistema in cui ciascun job è assegnato a uno specifico processore.

↳ Qui abbiamo due casi:

- 1) L'insieme di job nel sistema è predeterminato (= fissato in fase di progettazione)
- 2) L'insieme di job nel sistema è non predeterminato (= fissato "dynamically" successivamente, ma comunque non da parte dello scheduler).

Definizione:

SISTEMA REAL-TIME DINAMICO = sistema in cui lo scheduler può assegnare DYNAMICAMENTE un job a un qualunque processore disponibile.

↳ Qui abbiamo tre casi:

- 1) I job non sono interrompibili.
- 2) I job sono interrompibili ma non migrabili.
- 3) I job sono migrabili e interrompibili. → NB: la migrazione può essere più costosa di un semplice cambio di contesto.

→ MAKE SPAN = istante in cui termina l'ultimo job in un sistema multi-processore (nel caso in cui i job sono calcolati in fase).

↳ OCCHIO ALLE ANOMALIE DI SCHEDULAZIONE: è possibile che, se i job sono interrompibili e migrabili, il make span peggiori rispetto al caso di job non interrompibili.

Problemi dei sistemi statici:

→ Se un job va in OVERRUN (→ impiega + tempo del previs. nell'esecuzione), penalizzati solo i job dello stesso processore (almeno potentialmente).

→ Non ci sono costi di migrazione.

→ Sono + predibili.

→ Lo scheduler è + semplice.

Problemi dei sistemi dinamici:

→ Favoriscono il load balancing.

→ Hanno meno cambi di contesto e interruzioni di job.

→ Se un job impiega meno tempo del caso base, il tempo quadrato può essere potenziabilmente usato da tutti i task del sistema.

→ Se un job va in overrun, è più facile mitigare il problema sfruttando i processori liberi.

#### Algoritmi di schedulazione multi-processore:

→ Gli algoritmi clock-driven funzionano decentemente anche su sistemi multi-processore.

→ Per gli algoritmi priority-driven è un macello.

↓ 3 PROBLEMI

1) EFFICIENZA : effetto Dhall.

2) PREDECIBILITÀ COMPROMESSA: si hanno più anomalie di schedulazione.

3) TEST DI SCHEDULABILITÀ COMPRESOSSO: certi problemi non vengono più.

#### Teorema di Dhall: (PROBLEMA 1)

✓ Numero di processori  $M \geq 2$ , I任务 di task con utilizzazione bassa che non sono schedulabili con RR, RM o EDF.

#### Risultato confortante:

L'effetto Dhall si verifica solo se  $\exists$  task con un'utilizzazione molto alta.

#### Anomalie di schedulazione: (PROBLEMA 2)

Ecco, abbiamo una nuova anomalia di schedulazione: l'aumento del numero di processori.

↳ Solo nel caso di sistema statico con job interrumpibili non si soffre di anomalie.

#### Teorema di Lauerac, Melkum e Massé: (PROBLEMA 3)

Usando uno scheduler globale a priorità fissa a livello di task, l'istante in cui un job del task  $T_i$  è rilasciato contemporaneamente ai job di tutti i task di priorità superiore  $T_1, \dots, T_{i-1}$  non è necessariamente un istante critico di  $T_i$ .

Il teorema degli istanti critici valido per il caso uniprocessore non vale più!

#### Teorema di Oh e Baker:

Dato un sistema di task periodici con scadenze uguali ai periodi e  $M$  processori, se

è un qualsiasi algoritmo di scheduling partizionato (statico) con priorità fissa a livello di task:

$$U_x \leq \frac{m+1}{1+2^{-1/m}}$$

→ da scheduler cerca una partizione dei task, nel momento in cui questi vengono relasciati.  
↳ SICURAMENTE NON ABBIANO MIGRAZIONI

2) In realtà il teorema è stato migliorato:

$U_x$  non può superare  $\frac{m+1}{2}$  per tutti gli algoritmi di scheduling partizionato e per tutti gli algoritmi con priorità fissa a livello di job.

3) Corollario:

EDF, nel caso multiprocessore, non può essere ottimale.

Comunque sia, i<sup>l</sup> algoritmi con priorità dinamica a livello di job che sono ottimali anche nel caso multiprocessore, a fatto che stanno attivare (cioè hanno perfettamente idea di quali job verranno rilasciati in futuro e in quali istanti di tempo).

4) Una classe di algoritmi ottimali nel multi-processore è la classe FAIR, secondo cui:  
• ogni task ottiene una fetta di processore proporzionale alla sua dimensione ( $\rightarrow$  l'assegnaz. si però fatta ogni tot quanti di tempo).

5) Sono algoritmi costosissimi da implementare: non vengono tipicam. usati nell'ambito real-time.

Schedulazione partizionata:

Un algoritmo di schedulaz. partizionata ha due componenti:

→ Allocazione dei Task ( $\rightarrow$  calcolo del numero di processori necessario + assegnaz. di ogni task a un processore specifico) → PROBLEMA NP-HARD

→ Problema di priorità.

6) Nella pratica, vengono usati algoritmi per l'allocatione dei task che forniscono delle soluzioni "approssimate" (non ottime).

Per stabilire la bontà di questi algoritmi, si può ricorrere a 3 metriche:

- RAPPORTO DI APPROSSIMAZIONE
- FATTORE DI ACCELERAZIONE
- FATTORE DI UTILIZZAZIONE

### Algoritmo RMFF (Rate Monotonic First Fit):

- Ordino i task e li ordino per periodi non decrescenti.
  - Ordino arbitrariamente i processori.
  - Proffro  $T_1$  su  $P_1$ ; per quanto riguarda  $T_2$ , verifico se c'entra in  $P_1$ : se no, loaprofro in  $P_2$ ; e così via.
- E' chiaro che così si ottiene un numero di processori che non è pari al minimo teorico.
- $U_{RMFF} = m(\sqrt{2}-1)$  → circa il 42% dei processori può essere utilizzato (al più).
  - Fattore di approssimazione = 2,23 → l'algoritmo prevede 2,23 n processori, dove n è l'ottimale.
  - Non possiamo usare RMFF come algoritmo on-line (in cui sappiamo solo durante l'esecuzione quali job arrivano).

### Algoritmo FFDU (First Fit Decreasing Utilization):

È come RMFF, solo che i task vengono ordinati non sul periodo, bensì per fattori di utilizzazione decrescenti.

- $U_{FFDU} = m(\sqrt{2}-1)$
- Fattore di approssimazione = 1,67
- Non possiamo usare FFDU come algoritmo on-line.

### Algoritmo RM-FF:

È come gli altri due, solo che i task non vengono ordinati all'inizio.

- $U_{RM-FF} = m(\sqrt{2}-1)$
- Fattore di approssimazione = 2,23
- Possiamo usare RM-FF come algoritmo on-line.

### Algoritmo EDF-FF:

Prendiamo RM-FF e usiamo EDF al posto di RH: ecco EDF-FF.

$$U_{EDF-FF} = \frac{\beta m + 1}{\beta + 1} \quad \text{dove } \beta = \left\lceil \max_k \frac{e_k}{p_k} \right\rceil$$

Fattore di approssimazione = 1,7.

Guardando  $U_{EDF-FF}$ , notiamo che, per  $\beta \rightarrow +\infty$  ( $\equiv$  caso in cui abbiamo solo task picco = massimi), si può raggiungere un'utilizzazione del 100%.

20/12/2022

Tocca ancora attivare il nostro job periodico: lo facciamo in show ticks (E' possibile (così non abbiamo nulla di effettivamente periodico ma vabbè, è solo una prova))  
→ (Cioè, stiamo schedulando il job periodico artificialmente :))

→ FUNZIONA, YAY!

22/12/2022

Algoritmo EDF-FDD:

E' un'estensione degli algoritmi basati su "first fit" anche al caso dei task sporadici con scadenze arbitrarie.

Schedulare multiprocessore globali (non coordinati).

Assumiamo che i job siano tutti INDEPENDENT, INTERROGANI, MIGRABILI e SENZA AUTO-SOSPENSIONE.

Teorema:

Se scheduliamo i task (indip., interrapp., migrabili) con algoritmi a priorità FISSA, allora a che punto con sistemi PREDECESSORI e quindi non presentano anomalia sul tempo di esecuzione dei job.

→ TUTTROPPO LE ALTRIE RICHIERECCHE CONTINUANDO A ESISTERE.  
↳ By the way, la grossa del problema l'abbiamo risolto.

→ Gli algoritmi globali hanno senso di esistere nel momento in cui, nel caso uniprocessore e nel caso parzionale, il meglio lo ottengono con EDF (o EDF-FF) che è un algoritmo statico a livello di job → mentre invece con gli algoritmi globali riusciamo a fare qualcosa in più.

Algoritmo EDF globale:

Un sistema di task periodici con scadenze implicite, utilizzar. tabell. Ut e utilizzat. max dei task  $U_{max}$  è schedulabile con EDF globale su un sistema con m processori se:

$$U_t \leq m - (m-1) U_{max}$$

↳ per  $U_{max} \rightarrow n$  si va incontro all'effetto Dhall.  
per  $U_{max} \rightarrow 0$  EDF globale è ottimale.

↳ Se sostituiamo l'utilizzazione con la densità, il teorema vale anche per i task con scadenze non implicite.

È possibile avere algoritmi di scheduling con priorità fissa migliori di EDF globale?  
Sì, se mescoliamo gli algoritmi a priorità fissa di task e gli algoritmi a priorità fissa di job: otteniamo così degli algoritmi ibridi.

#### Algoritmo EDF-US [ $\zeta$ ]:

- $\zeta$  è un parametro dell'algoritmo ed è  $\leq 1$ .
- Una parte dei task ha priorità fissa: sono i task  $T_i$  t.c.  $e_i > \zeta$ , e hanno tutti la stessa identica priorità.
- La parte rimanente dei task ha priorità inferiore e viene schedulata con EDF.
- Ai task "grandi" (con priorità maggiore) sono assegnati per primi i processori disponibili.  
↳ In tal modo siamo prevenendo l'effetto Dhall.

#### Teorema:

Un sistema di task spadino con scadenze implicate è schedutabile con EDF-US [ $m/(2m-2)$ ]  
su  $m$  processori se  $U_f \leq m^2/(2m-2)$  →  $U_f$  è meglio di  $(m+1)/2$

#### Corollario:

Con EDF-US [ $\frac{1}{2}$ ] abbiamo  $U_f \leq \frac{m+1}{2}$

Questo sembra essere ex mejo per gli algoritmi di scheduling globale a priorità fissa.

#### ATTENZIONE

Esistono algoritmi con la stessa soglia ( $\frac{m+1}{2}$ ) che sono in grado di schedulare insiemi di job non schedutabili con EDF-US [ $\frac{1}{2}$ ].

#### Algoritmo EDF(K):

AD ESEMPIO

- $K$  è un parametro  $< m$ .
- Una parte dei task ha priorità fissa: sono i  $K-1$  task  $T_i$  che hanno fattore di utilizzazione  $e_i$  più alto.
- Per il resto, è uguale a EDF-US [ $\zeta$ ].

#### Teorema:

Un sistema di task spadino con scadenze implicate è schedutabile con EDF(K) su  $m$  processori se  $(K-1) + \lceil \frac{U_f + M_k}{1 - M_k} \rceil \leq m$ , dove  $M_k =$  fattore di utilizzazione del  $k$ -esimo task non a priorità fissa.

### Scheduler Real Time:

Questo non è facile.

#### Teorema:

Un sistema di task periodici con scadenze implicate f.c. ( $U_{max} < \frac{m}{3^{m-2}}$ ) è schedulabile solamente con RM su m processori se:  $U_i < \frac{m^2}{3^{m-1}}$ .  $\rightarrow$  SO PEGGIORANDO RISPETTO A EDF ALCUNI

#### Algoritmo RM-US [ξ]:

È un algoritmo simile del tutto analogo a EDF-US [ξ], con l'unica differenza per cui i task con priorità minima fissa vengono schedulati con RM anziché con EDF.

$\hookrightarrow$  Anche qui le prestazioni sono peggiori rispetto all'algoritmo EDF-US [ξ].

Inoltre, così come abbiamo definito EDF(K), possiamo definire RM(K).

#### GROSSO PROBLEMA NELLA PRATICA:

$\rightarrow$  Non riusciamo a fare in modo che i job siano indipendenti fra loro. E, se viewiamo maleare questa ipotesi, nei sistemi multi-processore tutti i risultati teorici che abbiamo analizzato vengono meno!

CONSEGUENZA: Nessun ente certificatore certificherà mai un sistema real-time mult-processore.

10/01/2023

### SISTEMI OPERATORI REAL-TIME

OBETTIVI DI UN SO: far funzionare i dispositivi, controllare l'uso delle risorse hardware. Sch. costituire un'astrazione dell'architettura fisica, distribuire le risorse hw alle applicazioni! Tu implementa i servizi di comunicazione, ... deve fare 1000 cose.

$\rightarrow$  I SO real-time differiscono da quelli tradizionali nelle applicazioni che vi girano sopra, che sono applicazioni che devono rispettare determinati vincoli temporali. Inoltre, i sistemi real-time sono anche embedded, per cui devono essere compatti, scalabili e con scialto consumo delle risorse.

Molti SO per sistemi embedded / real-time sono basati sul ~~modello~~ a microkernel, dove il microkernel è un piccolo programma che realizza pochi servizi essenziali (scheduling e comunicazione ha processi); gli altri servizi sono sì implementati nel SO ma non sono privilegiati. Il vero vantaggio del microkernel sta nel fatto che è piccolo, per cui è facile esaminarne il codice con lo scopo di certificare i sistemi real-time.

MA i microkernel hanno anche un grosso svantaggio: sono LENTI.

→ Nei SO i Kernel monolitici (come Linux) prevedono moltissimi servizi ~~per~~ che devono girare in modalità privilegiata, i microkernel ne hanno pochi (come la comunicazione tra processi): di conseguenza, per determinate operazioni (come comunicazione cd file system, che è gestito da un processo di livello user), le prestazioni calano.

→ I SO come Windows e MAC-OS sono considerati KERNEL LIQUIDI, che sono nati come microkernel ma poi ci si è resi conto che non ~~poteva~~ poteva essere, per cui in fase di sviluppo si è tentato di aggiungere servizi al Kernel.

con conseguenza: non hanno né i vantaggi dei ~~microkernel~~ microkernel, né i vantaggi del Kernel nel monolitico.

→ I RTOS (Real-Time Operating System) non hanno come priorità l'efficienza del Kernel, ed è per questo che sono basati sul modello a microkernel.

Inoltre, i requisiti esatti che i RTOS devono soddisfare dipendono dall'applicazione. In ogni caso il SO deve gestire le interruzioni in modo rapido, altrimenti le periferiche potrebbero non funzionare bene.

Schedulazione nei RTOS:

Tutti i RTOS implementano politiche di scheduling preemptive con priorità fissa.

In molti RTOS lo scheduler è esclusivamente clock-driven.

Generalmente i RTOS offrono un numero adeguato di livelli di priorità, dell'ordine delle centinaia.

MA → In generale un numero finito di livelli di priorità causa una perdita di schedulabilità, in particolare quando il numero di livelli ASSEGNATI ( $\rightarrow$  previsti a livello periodico) supera il numero di livelli supportati dal sistema.

Di fatto, in tal caso bisogna mappare le priorità teoriche ~~sui~~ quelle supportate in modo tale che alcune priorità teoriche collassino su una stessa unità fisica (supportata dal sistema).

→ PROBLEMA: come tener conto nell'analisi di schedulabilità che alcuni task sono altre priorità iden~~tic~~he? → BISOGNA considerare le formule riportate sulle slide.

È chiaro che comunque le funzioni di tempo necessario crescono, per cui le prestazioni del sistema peggiorano ( $\rightarrow$  diventano + propensi a non rispettare le scadenze) in particolar modo se utilizziamo un'associazione LINEARE tra priorità assegnate e priorità di sistema.

Per mitigare i danni, conviene conservare il + possibile la distinzione tra i livelli di priorità più alta.

Teorema:

Per l'algoritmo RM, con scadenze pari al periodo e numero  $m$  di task alto, dando l'associazione a supporto costante con  $g = \min_{1 \leq k \leq m} g_k$ : allora:

$$U_{RM}(g) = \begin{cases} \ln(2g) + 1 - g & \text{se } g \geq 1/2 \\ g & \text{se } g \leq 1/2 \end{cases}$$

dove  $U_{RM}(g)/\ln 2$  misura la probabilità di schedulabilità.

Sono ancora pochi i RTOS che supportano matematicamente una politica di scheduling.

## PARTE FINALE DI SERT

10/01/2023 (SECONDA PARTE)

### Standard per RTOS:

Hanno un ruolo importante perché consentono:

- La portabilità delle applicazioni.
- L'interoperabilità dei sistemi.
- La possibilità di implementazioni alternative dei RTOS.

vediamo qualche esempio di standard.

### POSIX:

È un'estensione di UNIX per i sistemi real-time. Definisce API per ottenere:

- Multitasking.
- Memoria condivisa.
- Code di messaggi a priorità.
- Schedulazione preemptiva a priorità fissa.
- Servizi sporadici.

E' molto ATTO

Lo standard definisce 4 profili differenti:

- 1) Minimal Real-Time System (PSE51).
- 2) Real-Time Controller (PSE52).
- 3) Dedicated Real-Time System (PSE53).
- 4) Multi-purpose Real-Time System (PSE54).

NON TUTTI I SO REAL-TIME DOVONO  
IMPLEMENTARE TUTTI I PROFILI.

### OSEK/VDX:

È lo standard creato in ambito automobilistico. Definisce API per un sistema real-time (OSEK) integrato in un sistema di gestione di rete (VDX).

È un sistema molto flessibile perché è proprio di molti produttori con esigenze diverse.

### ARINC 653 / APEX:

È uno standard usato soprattutto in ambito avionico. Fa uso di un insieme di API

chiamato APEX ~~per inter~~ che serve alle applicazioni per interagire con l'interfaccia ARINC653.

#### ITRON / μITRON:

In Europa è uno standard sconosciuto ma non lo è nel mondo asiatico: in Asia è utilizzatissimo per telefoni e tablet.

μITRON sta per Industrial TRON.

μITRON è una variante di ITRON per sistemi embedded a 8 o 16 bit.

? MA CI VARIANTI  
CE NE SONO UNA  
FRACCA

Ma cos'è TRON? È un'idea di progetto per i sistemi Real-Time proposta da un prof di un'università di Tokyo per motivi didattici.

→ Perché tale standard non è sbarcato in occidente? Perché, per programmare sistemi TRON, bisogna definire un'interfaccia grafica comprendente ideogrammi cinesi/giapponesi, per cui è reba incomprensibile per noi.

Caratteristiche comuni dei principali RTOS:

- Corrispondenza agli standard.
- Modularità/scalabilità.
- Codice basso radotto (poiché i RTOS sono basati su microkernel).
- Almeno 32 livelli di priorità supportati.
- Non supporto nativo di politiche di scheduling a priorità dinamica come EDF (almeno no generalmente).
- Possibilità di cambiare priorità a run-time.
- Scarso utilizzo di memoria virtuale, che tipicamente non serve e genera solo overhead, compromettendo la prevedibilità.

Ma quanti sono i RTOS? "NA FRACCA FRA": soprattutto all'inizio ognuno si scriveva il proprio.

Vediamo al volissimo qualche esempio.

#### VxWorks:

È un RTOS commerciale prodotto dalla Wind River: di fatto, è basato su microkernel Wind

### LYNxDOS:

È un SO commerciale prodotto dalla LynuxWorks.

### QNX Neutrino:

È un RTOS commerciale usato in ambito automobilistico e conforme allo standard POSIX.

È prodotto da QNX Software Systems.

### eCos:

È un RTOS open-source (gratis) che offre API compatibili con POSIX e µITRON.

Qui il Kernel è solo un package del sistema, e non deve essere per forza installato affinché il RTOS funzioni ( $\rightarrow$  può essere bare-metal).

### FREERTOS:

È un SO open-source la cui filosofia è avere il Kernel + piccolo possibile (fino a 9 KB di memoria).

### WINDOWS EMBEDDED:

È un SO per sistemi embedded ma NON per sistemi real-time; a differenza di Windows, ~~non~~ prevede un Kernel monolitico. Altro problema: supporta solo architetture mono-processore.

### ZEPHYR:

È un progetto che sta prendendo piede rapidissimamente, anche nell'IoT.

È un sistema completo e complicatissimo.

12/01/2023

### Oscilloscopio:

È uno strumento per misurare l'andamento di segnali elettrici nel tempo (e in particolare l'andamento delle tensioni).

Ci sono due ingressi (relativi a due ~~segnali~~ segnali) che vengono campionati; i valori campionati vengono mostrati su schermo.

Noi vogliamo far uscire il segnale elettrico dalla nostra Bob per farlo acquisire all'osciloscopio. Ad tale scopo possiamo utilizzare le linee GPIO (a cui i led sono collegati). In particolare, esistono due connettori per tirare fuori segnali.

Six. Nonostante abbiamo due soli ingressi, utilizziamo quattro linee: due linee sono sempre basse per dare un riferimento del valore di "terra", mentre sono le altre due linee alte a costituire il segnale.

2to → SEGNALE DI BASE: due onde contrapposte (quando una è bassa, l'altra è alta e viceversa).  
→ Se mi vogliano mostrare l'heartbeat (il nostro heartbeat misura 1 secondo, ricordi?), dobbiamo costruire una forma d'onda lunga 1000 tick ( $\equiv$  1000 ms).  
→ È possibile misurare la lunghezza d'onda posizionando i due cursori temporali sui due fronti d'onda (l'inizio e la fine dell'onda).

di → Nel nostro caso, la lunghezza d'onda è troppo bassa (sai 770 ms): questo è un problema dell'hardware, perché significa che le interruzioni arrivano troppo spesso. In particolare, abbiamo implementato il meccanismo di misurazione del tempo sul timer 0, che è troppo sensibile alle variazioni di temperatura ed è quindi troppo impreciso per i nostri scopi.

Per questo, dobbiamo usare TIMER-1 ms, un timer basato non più su un circuito RC, bensì su un circuito a QUARZO, un materiale <sup>così</sup> less sensibile alle variazioni di temperatura.

↳ Qui possiamo sfruttare il meccanismo dell'ADJUSTMENT, in cui alcuni tick durano un millesimo di ms e altri tick durano un millesimo di meno di 1 ms ma, mediamente, ci avviciniamo molto al millisecondo. Tale meccanismo va bene se vogliamo misurare il + accuratamente possibile il tempo reale ma non ci interessa molto la durata dei singoli tick; se questo va a va bene, disattiviamo l'adjustment.

TRIGGER = fisso la forma d'onda sullo schermo. <sup>dell'onda, nel punto</sup> Dalle inizio a distinguere l'onda stessa (è un punto di accorgaggio).

Fissato questo, è possibile misurare la variabilità massima del tick quando ADJUSTMENT = 1.



17/01/2023

### Linux in ambito real-time:

Linux, nonostante sia nato come un SO general purpose, ad oggi può essere considerato un SO real-time, anche se non è un sistema hard real-time. In particolare, il limite già per l'uso in hard-real time sta nella PREDESTABILITÀ; infatti:

- Il Kernel non è completamente interruttibile: ad esempio, le interruzioni nelle sezioni critiche sono disabilitate e, se tali sezioni critiche durano molto, le interruzioni possono rimanere appese per + tempo di quanto desiderato per un sistema real-time?
- Le interruzioni non vengono servite con priorità.

Il Kernel Linux è monolitico; tuttavia, ha senso considerarlo per i sistemi embedded, dato che è possibile configurare il Kernel in modo da ridurre drasticamente la sua dimensione.  
↳ ~~Il 72% dei sistemi embedded sviluppati nel 2020 usava Linux.~~

Come accennato prima, molti problemi legati all'uso di Linux in ambito real-time sono dovuti alla gestione degli interrupt, i quali hanno una priorità + elevata rispetto ai processi (anche se di per sé non hanno un valore che identifichi il loro livello di priorità esatto).

↳ Infatti, quando un interrupt  $I_1$  viene interrotto da un'altra interruzione  $I_2$ ,  $I_2$  sa che  $I_1$  nella schedulazione, anche se  $I_1$  avrebbe dovuto essere + prioritario ( $\rightarrow$  è + critico).

In Linux, i task real-time vengono schedulati in modo (quasi\*) all'atto separato rispetto ai task non-real-time. Inoltre, i task real-time hanno tre livelli di priorità:  
se finché abbiano fino a tre task real-time altri, vengono a schedularli secondo una politica real-time (priorità fissa); altrimenti, si possono avere alcuni task con la medesima priorità e questi vengono schedulati con una politica FIFO o Round-robin.

\* Quasi perché esistono casi in cui si può avere un'interruzione di priorità tra un task non real-time e un task real-time.

→ In Linux non è stato implementato l'algoritmo EDF puro perché è un SO che non tiene conto delle scadenze assolute dei processi.

#### PROBLEMI ANCORA DA RISOLVERE SULLA SCHEDULAZIONE:

1) Un processo real-time prioritario può essere interrotto da interrupt.

2) Linux implementa meccanismi di bilanciamento del carico sui sistemi multi-processore: la migration riduce la predicitività del sistema. → Per mitigare il problema, è possibile partitionare i processi sulle CPU in modo statico.

Come possiamo usare Linux in ambito hard-real time? Abbiamo due strade:

→ APPROCCIO MONO-KERNEL.

→ APPROCCIO DUAL-KERNEL.

#### Approccio mono-Kernel:

Consiste nel modificare radicalmente il Kernel in modo da:

→ Ottenere la quasi completa interoperabilità del Kernel.

→ Consentire una gestione prevedibile degli interrupt.

→ Implementare meccanismi di priority inheritance.

#### Approccio dual-Kernel:

Consiste nell'affiancare il nostro Kernel Linux a un altro Kernel che si occupi dei task real-time; qui il nostro Kernel Linux deve essere modificato molto meno.

Quello che si fa è introdurre uno strato intermedio fra l'hw e i SO: si tratta dell'IRQ Manager, il cui scopo è virtualizzare le interruzioni: in particolare, smista gli interrupt tra il RTOS e Linux.

Linux è attivo solo quando non ci sono processi real-time eseguibili.

#### VANTAGGI DELL'APPROCCIO MONO-KERNEL:

- Il Kernel, anche se stravolto, rimane quello, per cui le applicazioni real-time rimangono applicazioni Linux.

### VANTAGGI DELL'APPROCCIO DUAL-KERNEL:

- Richiede di modificare di meno il Kernel.
- È più facile certificare il sistema grazie alla presenza del RTOS.
- Le prestazioni dei task real-time non dipendono da Linux, per cui sono migliori.

### : RTAI (Real-Time Application Interface):

È un RTOS hard real-time con approccio dual-Kernel.

ADEOS (Adaptive Domain Environment for Operating Systems) è un progetto indipendente (ora abbandonato) da cui deriva il separaten Kernel di RTAI.

- Sia ADEOS che RTAI sono implementati come semplici moduli del kernel Linux.

Lo scheduler in RTAI è sia clock-driven che event-driven. Inoltre, è possibile migrare dei processi dal SO Linux al RTOS (a tal punto i processi non possono + invocare le API Linux). Chiaramente, è anche possibile avere processi nativamente RTAI e processi con alcun thread RTAI e altri ~~non~~ thread Linux.

### Linux PREEMPT\_RT:

È il principale progetto open-source per realizzare un RTOS mono-Kernel basato su Linux. Si tratta di una patch di Linux (che si evolve + lentamente di Linux stesso: ad esempio, non esiste ancora la patch per il kernel 6.1).

Ha due obiettivi:

- 1) ME LO SO' PERSO
  - 2) ME LO SO' PERSO PURA
- } solo slide

Comunque sia, l'idea fondamentale della patch PREEMPT\_RT consiste nella sostituzione di alcuni meccanismi di sincronizzazione. Per giunta, cerca di mitigare il problema dato del bilanciamento del carico mettendo la migrazione dei task correntemente in CPU e facendo modo di distribuirsi egualmente sulle CPU i task ad alta priorità.