Indice

Contents

1 Introduzione

L'idea alla base di MrsP prende spunto da molteplici approcci tipici dei sistemi real-time single e multi processor che presi singolarmente non portano a grandi benefici, ma con la combinazione esposta in questo documento si ottiene un protocollo innovativo che inoltre porta con se' il vantaggio di poter utilizzare tecniche di analisi di schedulatibilita' single-processor.

Nei prossimi capitoli viene proposta una carrellata dei protocolli e delle tecniche a cui si ispira MrsP, un modello concettuale ed infine una panoramica dell'implementazione. Dove ritenuto utile sono forniti dei grafici che simulano l'esecuzione del protocollo su determianti task set.

2 Protocololli di accesso a risorsa

Nei paragrafi seguenti alcuni protocolli esistenti vengono presi in considerazione come spunto per mettere in luce le caratteristiche desiderate in MrsP e mettere a confronto tra loro tecniche differenti con relativi vantaggi e svantaggi. Tali aspetti verranno poi ripresi ed argomentati nella descrizione del modello concettuale.

2.1 PCP/SRP

MrsP e' definito come un adattamento del protocollo per single-processor PCP/SRP a piattaforme multi-processor.

Un task che ottiene la risorsa innalza la propria priorita' al ceiling di risorsa, cioe' la priorita' piu' alta tra quelle dei task che la accedono. Questo permette al job di non essere prerilasciato dagli altri contendenti ad essa (oltre che da job di priorita' inferiore) e limitare cosi' il tempo di blocco.

Un job risulta bloccato se non riesce ad eseguire a causa di un job a priorita' inferiore che sta eseguendo ad un livello di priorita' superiore.

Il protocollo sopra menzionato porta con se' delle proprieta' che sono fondamentali per MrsP:

- un job subisce blocco solo una volta durante la sua esecuzione
- tale blocco avviene prima della sua effettiva esecuzione
- una volta che un job esegue, tutte le risorse a cui accede sono libere
- previene deadlock
- consente cumulo di risorse.

2.2 Sospensione ed attesa attiva

Nei protocolli che gestiscono l'accesso a risorsa tramite lock, un job in attesa di ottenerla puo' sospendersi o effettuare attesa attiva. Se in ambito single-processor la scelta obbligata e' la sospensione, nei sistemi multi-processor l'attesa attiva ha il vantaggio di limitare il tempo di blocco. Quest'ultima tecnica porta con se anche diversi svantaggi in quanto eseguire inibendo il prerilascio porta a soluzioni lontane dall'ottimale. Bisogna quindi decidere con accuratezza in che situazioni effettuare attesa attiva e con quali modalita'.

2.3 DPCP

Secondo il protocollo Distributed Priority Ceiling Protocols i task sono partizionati tra i processori e la risorsa grobale e' accessibile da ogni cpu. Nelle versioni precedenti le risorse globali sono assegnate ad un singolo processore che ne gestisce l'accesso, quindi i task per accederla devevano prima migrare al processore designato. La soluzione che propone DPCP permette di evitare le migrazioni per effettuare l'accesso.

2.4 MSRP

Il protocollo Multiprocessor Stack Resource Policy e' anch'esso un adattamento di SRP. Una risorsa non ha un unico ceiling, bensi' un ceiling per ogni processore essendo accessibile da ognuno di essi. Le richieste di accesso vengono gestite in ordine FIFO ed i job in attesa effettuano attesa attiva al ceiling della cpu di appartenenza.

2.5 OMLP

Questo protocollo gestisce l'accesso alla risorsa utilizzando una serie di accodamenti. Brevemente, un job innanzitutto deve contendere (tramite accodamento basato su priorita') con gli altri job della medesima cpu per ottenere il token del processore, una volta acquisito viene accodato per accedere alla risorsa globale.

Con OMLP viene messo in risalto il concetto di token per processore, in questo modo si limita ad uno il numero di job della stessa cpu che nel medesimo momento contendono la risorsa globale. In questo modo la coda per l'accesso alla risorsa risulta di lunghezza massima pari al numero di CPU.

2.6 Helping protocol

SPEEP e M-BWI, anche se in modo molto diverso tra di loro, introducono il concetto di helping protocol. Un job in attesa di ottenere la risorsa e' in grado di eseguire la sezione critica corrispondente per conto del lock holder. Grazie a questo approccio l'esecuzione della risorsa globale del job che viene prerilasciato durante la sezione critica viene portata a termine da uno dei job che stanno effettuando attesa attiva.

3 Analisi di schedulabilita' ed accesso alla risorsa

Come accennato in precedenza MrsP ha come obiettivo quello di proporre un protocollo di accesso a risorsa che permetta di utilizzare tecniche di analisi di schedulabilita' single processor ad un sistema multi processor.

Essendo il protoccollo un'estensione di PCP/SRP, prenderemo in analisi la Response-Time Analysis che incorpora tale protocollo:

$$R_i = C_i + B_i + I_i$$

Nell'equazione C_i rappresenta il WCET dell'i-esimo task sommato alle sezioni critiche corrispondenti alle risorse, I_i l'interferenza subita a causa dei task a priorita' superiore residenti nello stesso processore e B_i e' pari al tempo di blocco causato dalla presenza delle risorse.

L'ultimo termine rispecchia la natura della risorsa, se in ambito single-processor la serializzazione dell'accesso deriva dalla presenza di un'unica unita' di esecuzione, in sistemi multi-processor essa dev'essere gestita tramite protocollo di accesso. L'aumento del numero di processori porta con se' il vantaggio di diminuire l'interferenza tra i task, in quanto le possibili esecuzioni in parallelo aumentano, ma dall'altra parte allungano il tempo per ottenere la risorsa.

Alla luce di queste considerazioni, il modello che a breve verra' esposto e' conseguente alla necessita' di proporre un protocollo in grado di utilizzare la tecnica di analisi ad un sistema multi-processor partizionato incorporando nel tempo di blocco e nell'esecuzione delle risorse i tempi di accesso alla risorsa globale. Per fare questo dobbiamo quindi tenere conto del costo di serializzazione degli accessi.

3.1 Accessi in parallelo

Definiamo un fattore e che indichi il numero di potenziali accessi in parallelo, quindi non il numeri di task che la accedono, bensi' la quantita' di processori in cui e' allocato almeno uno di questi task. Quindi se tutti i task sono allocati nello stesso processore tale fattore sara' pari ad uno, se sono divisi tra due cpu il valore sara' due e cosi' via.

In MrsP ritroviamo una delle caratteristiche principali di PCP/SRP: un job che inizia ad eseguire solo se tutte le risorse che utilizza sono logicamente libere, con la differenza che il tempo di esecuzione della sezione critica della risorsa deve incorporare anche il tempo necessario per ottenerla data la contesa con le altre richieste dagli altri e processori. Nel caso peggiore la

richiesta viene accodata dietro a $e - 1$ richieste e per ottenere la risorsa dovremo di conseguenza attendere la sua esecuzione da parte di tutti i job. Data questa osservazione, il costo di esecuzione della risorsa sara' quindi pari alla lunghezza della sezione critica, che assumiamo identica per tutti i job, moltiplicata per il fattore e, cioe' la nostra esecuzione piu' quella degli altri job accodati prima della nostra richiesta.

I costi della serializzazione devono essere inseriti nell'equazione discussa nel paragrafo precedente, questo comporta quindi sostituire il tempo di esecuzione della risorsa con il nuovo valore sia nel calcolo della componente di blocco che nel costo di esecuzione del task stesso.

3.2 FIFO e spinning

Il fatto che un job esegua solamente quando tutte le sue risorse sono libere non e' un fattore sufficiente per rispettare l'equazione che vogliamo utilizzare per poter analizzare la schedulabilita' di un insieme di task.

Quello che vogliamo ottenere e' assicurare al job che ha effettuato la richiesta un tempo massimo, proporzionale al numero di accessi in parallelo, entro il quale accedere alla risorsa ed eseguire la propria sezione critica.

Se il job si sospende dopo aver accordato la propria richiesta potenzialmente si incorre in molte delle conseguenze che un sistema suspension-based comporta, quindi subire inversione di priorita', tempo di blocco aggiuntivo ed inoltre altre richieste verrebbero accodate.

In MrsP un job che effettua la richiesta deve attuare attesa attiva fino a che non venga garantito l'accesso in ordine FIFO. Questo comporta la necessita' di trovare un compromesso tra il tempo di attesa per ottenere la risorsa e l'influenza subita da parte dei task a priorita' superiore che non la accedono.

Una prima soluzione consiste nel disabilitare il prerilascio, quindi effettuare spinning alla priorita' piu' alta nel processore in cui e' allocato il job. In questo modo si garantisce un tempo di attesa minimo dato solamente dalla lunghezza della FIFO, ma d'altro canto si obbligano i task a priorita' maggiore alla propria che non accedono la risorsa a subire blocco. Un altro protocollo suggerisce di effettuare spinning ad un ceiling locale, in questo modo non si bloccano i task a priorita' superiore, ma si prolunga il tempo di blocco dei task a priorita' inferiore in caso di prerilascio.

MrsP adotta la seconda soluzione, anche se al momento non e' ancora sufficiente per rispettare l'equazione che ci siamo prefissati di utilizzare. Ri-

mandiamo tale problematica al paragrafo successivo, dove viene discusso il modello concettuale.

4 Modello concettuale

Il modello concettuale di MrsP deriva dalle considerazioni fatte finora e prende spunto dai protocolli precedentemente discussi per combinarli in modo da creare un protocollo che soddisfi i nostri obiettivi.

Il sistema di riferimento e' multi-processor partizionato con scheduler fixed priority, quindi ad ogni processore e' abbinata una coda in cui vengono accodati i job pronti per l'esecuzione con ordinamento in base alla loro priorita'. Lo scheduler quando necessario seleziona il job che deve eseguire dalla testa della coda.

Di seguito verranno introdotti dei task set utilizzati nel resto del documento per dimostrare i meccanismi che man mano verranno discussi.

4.1 Ceiling

Uno dei punti fondamentali visti finora e' garantire che un job quando inizia la propria esecuzione tutte le risorse di cui necessita siano libere e questo si ottiene innalzando la priorita' del job al ceiling della risorsa globale. E' cruciale quindi decidere come calcolare tale ceiling.

Alla risorsa globale sono assegnati un insieme di celing, uno per ogni processore in cui e' assegnato almeno un task che la accede. Per ogni processore tale valore e' pari alla priorita' massima tra tutti i task in essa allocati e viene calcolato offline.

Un job che accede alla risorsa quindi innalza la propria priorita' a tale valore, accoda la propria richiesta con ordinamento FIFO ed inizia ad effettuare attesa attiva. Una volta che la richiesta raggiunge la testa della coda viene garantito l'accesso ed il job esegue la sezione critica corrispondente alla risorsa.

Questo funzionamento oltre ad estendere le caratteristiche di PCP/SRP ci permette di garantire che la lunghezza della FIFO sia al massimo pari al numero di potenziali accessi in parallelo, fattore di cui abbiamo discusso in precedenza.

Task	Processor	Release time	Execution time
τ_1	P_1	0	3
τ_2	P_2	0	3
τ_3	P_3	0	3

Table 1: Ceiling, FIFO e spinning

Di seguito e' illustrato un sistema partizionato multi-processor in presenza di una risorsa globale gestita con protocollo d'accesso basato su MrsP con i meccanismi finora descritti.

Inizialmente il task set (tabella 1) e' cosi' composto:

- 3 processori: P_1, P_2 e P_3
- 3 task con il medesimo comportamento, ognuno accede alla medesima risorsa globale
- lunghezza sezione critica pari a 2 unita' di tempo per ogni task.

Si assume che ogni task acceda alla sezione critica alla fine della propria esecuzione del WCET, per esempio se un job ha un *Execution Time* pari a 3 ed una *Critical Section Lenght* di 2 esegue prima le 3 unita' di tempo per l'esecuzione locale e poi esegue la risorsa, per un totale di 5.

Nel grafico 1 i tre job hanno il medesimo release time, ogni processore e' libero ed iniziano quindi ad eseguire al tempo t. All'istante $t+3$ effettuano simultaneamente una richiesta di accesso alla risorsa globale, le relative priorita' vengono innalzate al ceiling che assumiamo calcolato secondo il modello finora descritto e le richieste accodate in ordinamento FIFO. Al tempo di $t+3$ nella coda vi sono le richieste di τ_3, τ_2 e τ_1 in questo preciso ordine. L'accesso viene quindi garantito al job con la richiesta in testa alla coda, mentre gli altri due effettuano attesa attiva. Al rilascio della risorsa, $t+5$, da parte di τ_3 la relativa richiesta viene rimossa dalla coda ed l'accesso viene garantito al nuovo job in base all'ordinamento FIFO, quindi τ_2 smette di effettuare attesa attiva ed inizia l'esecuzione della sezione critica. Al tempo $t+7$ il medesimo meccanismo rilascia la risorsa in favore di τ_1.

4.2 Prerilascio

Il modello allo stato attuale soffre di una pecca precedentemente discussa, se il job che sta eseguendo la sezione critica al ceiling di risorsa nel proprio processore viene prerilascio il tempo di blocco subito dagli altri job aumenta, ed inoltre aumenta il tempo necessario dei job che stanno effettuando attesa

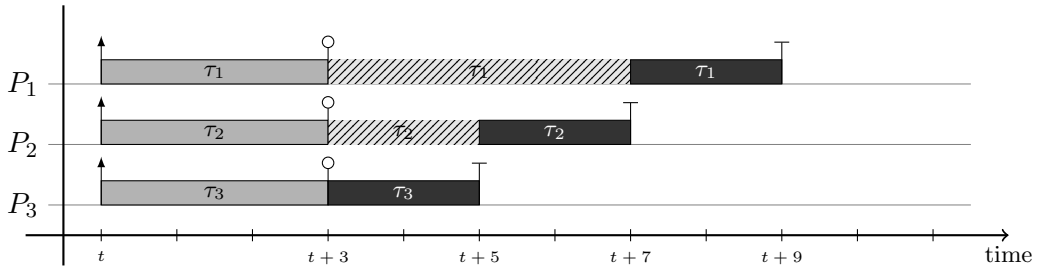

Figure 1: Ceiling, FIFO e spinning

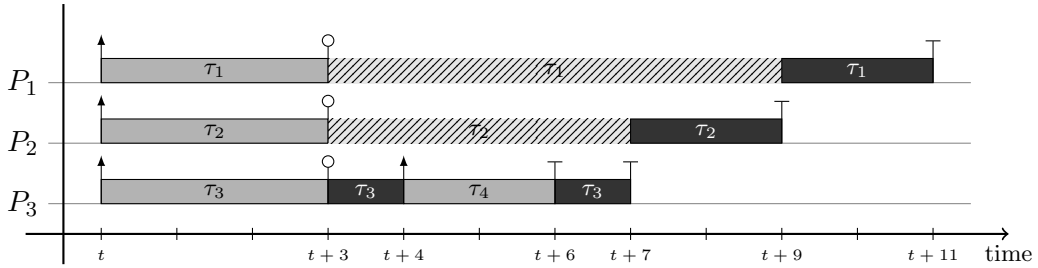

Figure 2: Prerilascio dannoso

attiva per ottenerla.

Modifichiamo il task set precedente aggiungendo un task a P_3, quest'ultimo ha priorita' superiore rispetto al ceiling del processore, quindi e' in grado di prerilasciare il job di τ_3. Il job di τ_4 ha release time al tempo $t + 4$ durante l'esecuzione della sezione critica da parte di τ_3 ed esegue per due unita' di tempo. In 2 si evidenzia come i tempi di accesso alla risorsa da parte degli altri job in attesa vengano allungati.

Di seguito viene descritto un aspetto del modello contettuale che parte dal presupposto che una sezione critica non e' altro che una serie di istruzioni a se stanti, e quindi eseguibili da qualunque job in modo indistinto.

La caratteristica innovativa di MrsP mira a risolvere tale problema, un job in attesa di ottenere la risorsa puo' proseguire l'esecuzione della sezione critica del job prerilasciato. Tale meccanismo viene effettuato scorrendo in ordine FIFO la lista dei job che hanno effettuato richiesta fino a trovarne uno in esecuzione che possa far proseguire la sezione critica per conto del lock holder, il quale trovera' il risultato della risorsa una volta che tornera' running nel proprio processore.

Una volta che un job ha eseguito la sezione critica per conto di un altro job,

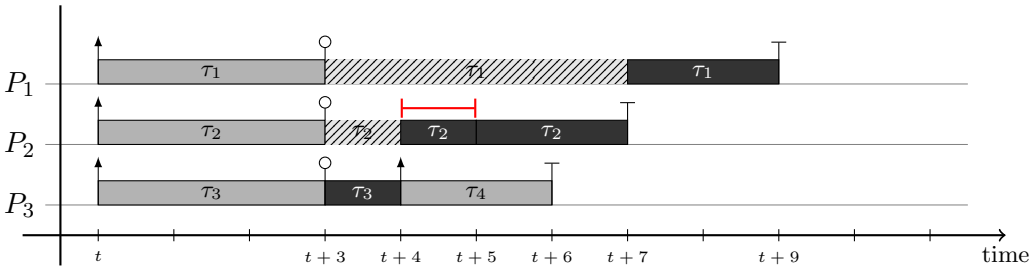

Figure 3: Il job in attesa prosegue la sezione critica del job prerilasciato

il meccanismo e' quello descritto in precedenza, cioe' la risorsa viene liberata ed ottiene la risorsa il job in testa alla coda, con la differenza di dover mettere a disposizione il risultato della risorsa in modo tale da consegnarla al job destinatario.

Nel grafico 3 viene rivisto il sistema precedente e si nota come i tempi di attesa rispettino il modello finora descritto nonostante il prerilascio subito da parte del lock holder. Al tempo $t + 4$ il job che detiene la risorsa viene prerilasciato in favore del job a priorita' piu' alta del ceiling. Entra in scena quindi il meccanismo appena enunciato, il job del task τ_2 viene selezionato da MrsP in quanto e' il primo ad essere disponibile secondo l'ordinamento della coda di richieste e si prende carico di portare a termine l'esecuzione della sezione critica della risorsa per conto di τ_3. All'istante $t + 5$ la risorsa viene rilasciata ed il risultato della sua esecuzione viene messo a disposizione di τ_3 che al tempo $t + 6$ lo legge e termina il proprio ciclo.

4.3 Gestione casi particolari

Il meccanismo descritto finora funziona nel caso ottimo, quindi in situazioni in cui al momento del prerilascio vi sia sempre un job che sta effettuando attesa attiva da incaricare di portare a termine l'esecuzione della sezione critica.

4.3.1 Caso pessimo

Nel caso in cui tale situazione non si verifichi, il job che detiene la risorsa e' destinato a riaccodarsi nella coda ready del proprio processore in attesa di tornare in esecuzione. Se durante il periodo di tempo in cui subisce interferenza nessun altro job effettua attesa attiva il modello finora descritto rispetta le caratteristiche che desideriamo in quanto nessun task ha dovuto attendere piu' del dovuto o rimanere bloccato piu' a lungo. In caso contrario MrsP necessita di un messanismo che notifichi che e' stata avvallata

Task	Processor	Release time	Execution time	Critical section length	Priority
τ_1	P_1	0	3	2	low
τ_2	P_1	4	2	0	high
τ_3	P_2	0	3	2	low
τ_4	P_2	4	1	0	high
τ_5	P_2	0	3	2	low
τ_6	P_2	4	2	0	high

Table 2: Gestione caso pessimo

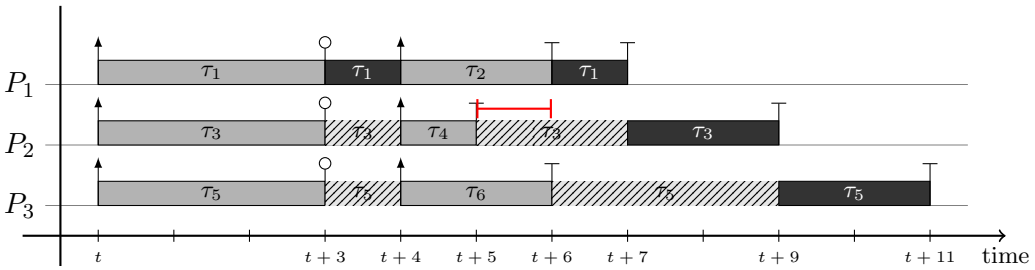

Figure 4: Task set 2 senza meccanismo di notifica

una nuova richiesta alla risorsa, e quindi aggiunta alla coda FIFO, oppure uno dei job in attesa e' tornato running e quindi disponibile per eseguire la sezione critica.

Con questo meccanismo assicuriamo che l'esecuzione della sezione critica possa progredire ogni volta che ve ne sia la possibilita', evitando quindi il prolungamento dei tempi per ottenere la risorsa e del tempo di blocco subito dai task a priorita' inferiore.

Modifichiamo il task set in modo tale da creare tale situazione (tabella 2), nei grafici 4 e 5 viene messo il risalto il funzionamento del modello senza e con il meccanismo appena annunciato.

I grafici sono per lo piu' identici ed il funzionamento e' quello atteso fino al tempo $t+4$: in tale istante la coda di MrsP presenta nell'ordine le richieste di τ_1, τ_3 e τ_5, ma al momento del prerilascio del job di τ_1 non vi e' nessun job che sta effettuando attesa attiva per ottenere la risorsa in quanto anch'essi sono stati prerilasciati nei corrispondenti processori. Il dententore della risorsa deve quindi riaccodarsi in attesa di poter nuovamente eseguire nella propria cpu.
All'istante $t + 5$ il job che sta effettuando attesa attiva in P_2 riprende ad eseguire, nel sistema senza meccanismo di notifica riprende lo spinning fino al rilascio della risorsa da parte del suo possessore (4), mentre con l'utilizzo

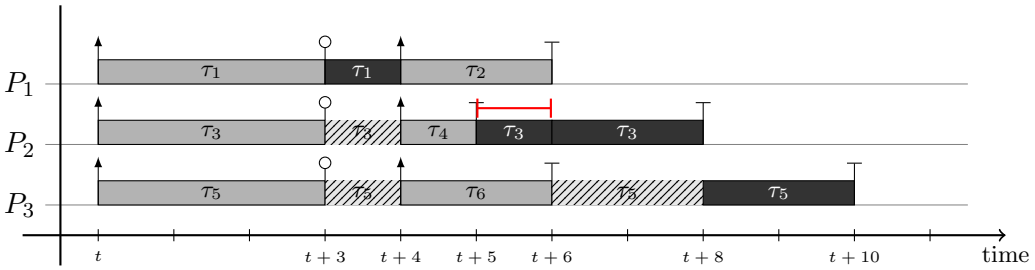

Figure 5: Task set 2 con meccanismo di notifica

della notifica MrsP si accorge che vi e' la possibilita' di far progredire la sezione critica interrotta (5).
Come ci si puo' aspettare i tempi di attesa sono ridotti ed il job di τ_1 trovera' il risultato della risorsa globale al proprio risveglio.

4.3.2 Caso intermedio

Un caso intermedio tra quello appena descritto ed il modello precedente e' quando il job che detiene la risorsa ritorna running mentre la sezione critica e' in esecuzione in un altro processore. In tale situziazione il lock holder riprende ad eseguire la sezione critica mentre l'altro job riprende ad effettaure attesa attiva. Se cosi' non fosse il processore del possessore della risorsa resterebbe inutilizzato, permettendo cosi' a job a priorita' inferiore al ceiling di eseguire. Come discusso in precedenza questa eventualita' e' dannosa per il protocollo in quanto non rispetterebbe le proprieta' di PCP/SRP che sono alla base di MrsP.

Anche in questo caso modifichiamo il task set finora utilizzato per mettere alla luce tale dinamica 6. Il task set rimane il medesimo eccezzion fatta per la sezione critica che per ogni task sara' pari a 3 unita' di tempo e non 2. Il comportamento e' quello appena descritto, al tempo $t + 6$ il job torna running in P_1 e riprende l'esecuzione della propria sezione critica mentre il job di τ_3 riprende l'attesa attiva.

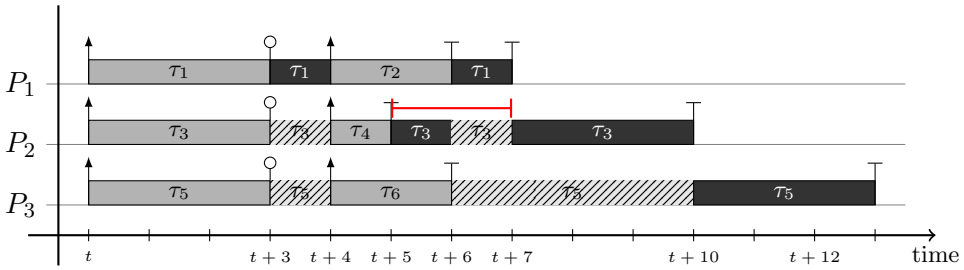

Figure 6: Task set 2 con sezione critica pari a 3 unita' di tempo.

5 Implementazione

Il modello concettuale discusso nel capitolo precedente non e' applicabile in fase di implementazione cosi' com'e'. In un sistema reale non e' possibile cedere l'esecuzione di una serie di istruzioni ad un altro job in quanto consisterebbe nel trasferire program counter e record di attivazione da un flusso di esecuzione ad un altro.

Questo fatto si traduce nel dover introdurre un approccio che sostituisca la possibilita' da parte di un job di eseguire la sezione critica per conto del possessore della risorsa senza pero' mutare i risultati ottenuti.

5.1 Il nuovo modello

La migrazione di un job da un processore ad un altro e' un meccanismo che torna utile nella nostra situazione anche se tipisco di sistemi con scheduler globale e non partizionato. Se nel modello concettuale il job che sta effettuando attesa attiva al posto di prendersi carico della sezione critica cede l'utilizzo del proprio processore al lock holder. Salvo qualche accorgimento che discuteremo in seguito il modello non subisce sostanziali variazioni.

Nei punti di seguito viene descritto il comportamento dei job secondo il nuovo modello:

1. un job esegue solamente se le risorse di cui necessita sono libere, una volta effettuata la richiesta di accesso essa verra' accodata in ordinamento FIFO nella coda della risorsa globale corrispondente, inoltre la sua priorita' viene innalzata al ceiling della risorsa globale relativa al processore in cui il task e' allocato
2. il job effettua attesa attiva fino a che non riesce ad ottenere la risorsa
3. una volta raggiunta la testa della coda e la risorsa e' libera ne ottiene l'accesso

4. in caso di prerilascio durante l'esecuzione della sezione critica il job migra in uno dei processori in cui si trova un job che sta effettuando attesa attiva per ottenere la medesima risorsa. Nell'attuare tale meccanismo il job che detiene la risorsa imposta la prioria priorita' pari al livello direttamente superiore al ceiling del processore in cui sta migrando, in questo modo il job che sta effettuando attesa attiva verra' prerilasciato in suo favore

5. nel caso in cui il job non stia eseguendo, nonostante possegga la risorsa globale, a causa di prerilascio in circostanze in cui non vi erano job che stavano effettuando attesa attiva, il meccanismo di notifica discusso in precedenza notifichera' a MrsP che un processore e' nuovamente disponibile alla migrazione, la quale avverra' secondo gli accorgimenti descritti al punto precedente

6. una volta completata la sezione critica viene tolta la propria richiesta dalla testa della coda, rilasciata la risorsa, ripristinata la priorita' originale ed il job migra nel processore in cui il task e' allocato.

Alcune puntualizzazioni. Nel punto 3, una volta ottenuta la risorsa, la richiesta di accesso non viene rimossa in quanto utile per la fase di ricerca di un processore disponibile in cui migrare, ed essendo in testa si predilige una migrazione alla cpu di origine piuttosto che in una delle altre disponibili. Nel punto 4 si presuppone che il livello di priorita' superiore al ceiling non sia assegnato a nessun task per ogni processore. Al punto 6 la migrazione viene effettuata solamente se necessario, quindi solo se i meccanismi di MrsP hanno portato il job ad eseguire in un processore differente dal proprio.

5.2 I costi della migrazione

In un sistema reale la migrazione ha un costo di cui bisogna tenere conto in fase di implementazione. Se nel modello concettuale nella situazione descritta nel grafico 6 l'esecuzione della sezione critica torna al lock holder, la migrazione puo' essere gestita in modo diverso.

Con il task set della tabella 3 vediamo come si evolve l'esecuzione dei job senza gestire tale caso, quindi lasciando il job ad eseguire nel processore in cui e' migrato in precedenza senza ritornare in quello di orgine.

Nel grafico 7 si nota come in P_1 non vi sia alcuna gestione della migrazione da parte del job di τ_1, quest'ultimo termina la propria esecuzioone della sezione critica in P_2 permettendo cosi' al job a priorita' inferiore di eseguire. Questo e' dovuto al fatto che un job non puo' eseguire in due processori contemporaneamente. Al tempo $t + 5$ il job di τ_3 inizia quindi ad eseguire

Task	Processor	Release time	Execution time	Critical section length	Priority
τ_1	P_1	0	2	3	low
τ_2	P_1	3	2	0	high
τ_3	P_1	0	0	3	very low
τ_4	P_2	0	2	3	low
τ_5	P_2	3	1	0	high

Table 3: Conseguenze della migrazione

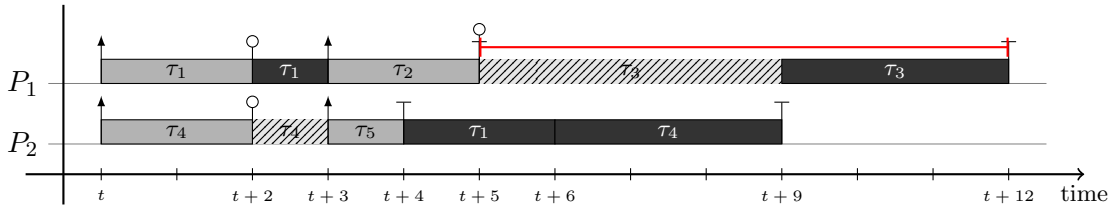

Figure 7: Task set 2 senza gestione della migrazione.

dopo che il job a priorita' piu' alta ha terminato ed effettua immediatamente la richiesta alla risorsa, ottenendo cosi' il ceiling a livello locale della risorsa globale ed accodandosi. In questo modo si violano i vincoli imposti dal protocollo secondo i quali ad un job e' permesso di eseguire solamente se le risorse di cui necessita secondo libere. Tra l'altro in questo modo ritarda il completamento dell'esecuzione di τ_1 in quanto non puo' tornare running al tempo $t + 6$ dato che un job con la stessa priorita' sta eseguendo e deve quindi attendere $t + 12$.

L'esempio appena visto risalta la necessita' di aggiungere qualche accorgimento per evitare che situazioni di questo tipo accadano. Le soluzioni possibili sono le seguenti:

1. eseguire un job che funga da placeholder fino al momento della migrazione al processore di origine
2. forzare il processore a rimanere inutilizzato fino al ritorno del job a meno di task a priorita' superiore al ceiling
3. eseguire la migrazione al processore di origine non appena possibile.

Le prime due soluzioni portano al medesimo risultato anche se la soluzione 1 comporta la gestione di un nuovo task mentre la seconda non fa altro che bloccare la coda di priorita' dello scheduler a meno delle circostanze appena citate.

Nel grafi 8 viene messa in risalto la seconda soluzione, in particolare al tempo $t + 5$ il job del task τ_3 non inizia eseguire nonostante il processore non sia

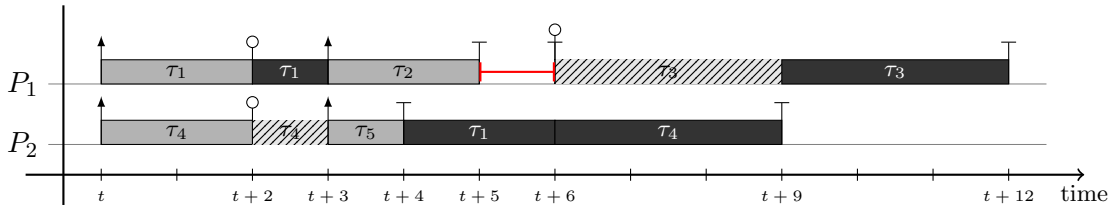

Figure 8: Gestione del processore inutilizzato tramite ceiling e blocco della coda ready.

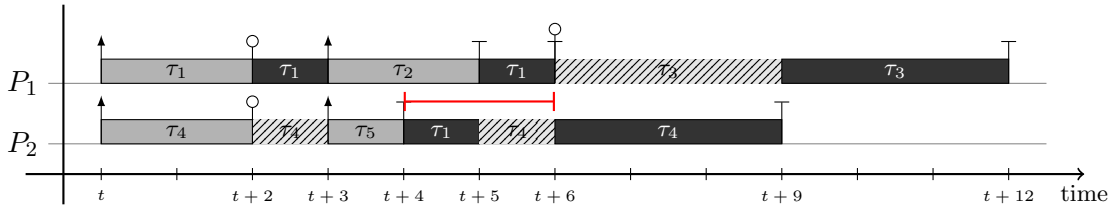

Figure 9: Migrazione al processore di origine non appena possibile.

utilizzato in quanto la sua priorita' non e' superiore al livello del ceiling. In 9 invece si nota come al tempo $t + 5$, non appena il job a priorita' piu' alta del ceiling finisce, venga effettuata la migrazione del lock holder.

5.3 Strutture dati

Per lo sviluppo del protocollo si e' reso necessario l'utilizzo di svariate strutture dati, alcune delle quali derivanti da LITMUS-RT mentre altre sono state create appositamente per la gestione di MrsP, in particolare della coda di richieste di accesso.

Struttura dati che rappresenta un nodo nella lista di job per l'accesso alla risorsa:

```
typedef struct queue_s {
  struct list_head next;
  struct task_struct* task;
} queue_t;
```

Risorsa globale MrsP, tiene traccia del lock holder corrente, dei task che vogliono accedere alla risorsa e le priorita' della risorsa per ogni CPU:

```
struct mrsp_semaphore {
        struct litmus_lock litmus_lock;
        spinlock_t lock;                    /* lock for mutual access to the struct */
```

```
        struct list_head task_queue;    /* tasks queue for resource access */
        struct task_struct *owner;      /* current resource holder */
        int prio_ceiling [NR_CPUS];     /* priority ceiling for each cpu*/
};
```

Per ogni cpu viene tenuto traccia del ceiling locale, ci permette di identificare se il valore di ceiling della risorsa globale del processore in questione e' in utilizzo o meno:

```
struct mrsp_state {
        int cpu_ceiling;
};
```

In presenza di un sistema partizionato, ogni CPU ha il proprio dominio:

```
typedef struct {
        rt_domain_t                 domain;
        struct fp_prio_queue        ready_queue;
        int                         cpu;
        struct task_struct*         scheduled;
        struct mrsp_semaphore*      sem;            /* global resource */
#define slock domain.ready_lock
} pfp_domain_t;
```

Oltre a quelle sopra elencate e' stata modificata la struttura del task in modo tale da poter tenere traccia di priorita' e cpu originarie per gestire le migrazioni.

5.4 Callback di LITMUS-RT

Scheduler Lo scheduler mette a disposizione delle callback per gestire gli eventi che caratterizzano l'esecuzione di un task e gli eventi di scheduling, in particolare per l'implementazione di MrsP sono due le funzioni modificate per adattare P-FP al protocollo.

```
static struct task_struct* pfp_schedule(struct task_struct * prev)
```

Funzione chiamata in ogni occasione in cui bisogna decidere quale job far eseguire sulla cpu corrente. Una volta determinato lo stato del job attualmente in esecuzione (prev), si decide se far eseguire o meno il job in testa alla ready_queue.

```
static void pfp_finish_switch (struct task_struct *prev)
```

Viene chiamata alla fine di ogni context switch per verificare se il job uscente deve migrare. Buona parte delle modifiche al normale flusso di P-FP sono state effettuate in questa funzione in quanto meno soggetta a vincoli di lock su cpu rispetto alla precedente funzione.

Risorsa globale All'interno della struttura mrsp_semaphore il campo dati litmus_lock rappresenta la risorsa a livello kernel, anche in questo caso vengono messe a disposizione delle callback per la gestione degli eventi:

```
int pfp_mrsp_open(struct litmus_lock* l, void* __user config)
```

Callback richiamata quando il primo job crea la risorsa o i successivi recuperano un riferimento ad essa. In questa fase di inizializzazione viene calcolato il ceiling della risorsa per ogni cpu.

```
int pfp_mrsp_unlock(struct litmus_lock* l)
```

Un job rilascia la risorsa.

```
int pfp_mrsp_lock(struct litmus_lock* l)
```

Un job tenta di acquisire la risorsa.

Funzioni aggiuntive Lo sviluppo del protocollo ha reso necessario fornire altre fuzionalita' oltre a quelle messe a disposizione da LITMUS.

Tra queste vi sono un insieme di funzioni che semplificano le migrazioni e la gestione della lista di task che accedono alla risorsa con ordinamento FIFO.

In particolare la funzione di ricerca scorre la lista alla ricerca di una cpu disponibile per eseguire il lock holder tra quelle in attesa di accesso alla risorsa. Una cpu per poter essere eleggibile deve essere libera, oppure il job attualmente in esecuzione deve avere priorita' maggiore o uguale al ceiling di risorsa.

5.5 Lock della risorsa

Un job tenta di acquisire la risorsa tramite la funzione pfp_mrsp_lock, il funzionamento rispetta il modello visto finora, in particoalre innalzamento della priorita' al ceiling locale della risorsa globale e spinning fino ad ottenere l'accesso.

```
int pfp_mrsp_lock(struct litmus_lock* l) {

struct task_struct* t = current;
struct mrsp_semaphore *sem = mrsp_from_lock(l);

/* 1. acquisisco il ceiling di risorsa nella cpu corrente */
t->rt_param.task_params.priority = (sem->prio_ceiling[get_partition(t)]);

/* 2. aggiorno il ceiling della cpu */
__get_cpu_var(mrsp_state).cpu_ceiling = (sem->prio_ceiling[get_partition(t)]);

/* 3. aggiungo il job alla lista per l'accesso alla risorsa */
queue_add_fifo(sem, t);

next = list_entry(sem->task_queue.next,queue_t,next);

/* 4. controllo se la risorsa e' libera e se il job e' in testa alla coda */
if(sem->owner == NULL && next->task == t) {

        /* 5. acquisisce la risorsa */
        sem->owner = t;

} else if(sem->owner != NULL) {

        /* 6. se la risorsa non e' libera controllo che il lock holder stia
              eseguendo */
        if(is_queued(sem->owner)) {
                owner = sem->owner;
                from_cpu = get_partition(owner);
                target_cpu = get_partition(t);
        }
}

/* 7. se il lock holder (J^lh) e' stato prerilasciato e non ha potuto migrare in quel
      determinato momento, gli cedo la cpu corrente */
if(owner) {

        /* 7.1 tolgo J^lh dalla coda ready in cui si trova
        *  7.2 imposto la priorita' di J^lh alla ceiling della cpu attuale -1
        *  7.3 aggiungo J^lh alla coda della cpu corrente
        *  7.4 richiamo schedule() per cedere l'esecuzione a J^lh */
        mrsp_dequeue_and_migrate(from_cpu, target_cpu, owner);
}

/* 8. Effettuo attesa fino ad ottenere la risorsa */
```

```
while(sem->owner != t)
{
        next = list_entry(sem->task_queue.next,queue_t,next);
        if(sem->owner == NULL && next->task == t) {
                sem->owner = t;
        }
}

return 0;
}
```

Nel punto 2 si tiene traccia dell'innalzamento di priorita' nella cpu, tale valore viene ripristano alla priorita' piu' bassa possibile al momento della release. Tale accorgimento e' stato discusso in precedenza, il valore di ceiling viene utilizzato per bloccare la coda ready del processore originario del job che detiene la risorsa nel caso in cui migri.

Il punto 7 e' parte del meccanismo di notifica di cui si serve MrsP per far eseguire il lock holder che a causa di determinati interleaving si trova accodato nella coda ready di un processore.

5.6 Rilascio della risorsa

In fase di rilascio della risorsa il job ripristina la propria priorita', se necessario migra al processore di origine e ristabilisce il valoce di ceiling della cpu.

```
int pfp_mrsp_unlock(struct litmus_lock* l)
{

struct task_struct *t = current;
struct mrsp_semaphore *sem = mrsp_from_lock(l);

/* 1. rilascio della risorsa */
sem->owner = NULL;

/* 2. ripristino la priorita' al suo valore iniziale */
t->rt_param.task_params.priority = t->rt_param.task_params.priority_for_restore;

/* 3. tolgo la testa della coda FIFO */
queue_pop(sem);

/* 4. se necessario, ritorno alla cpu di appartenenza */
mrsp_migrate_to_from_resource(t->rt_param.task_params.home);

/* 5. ripristino il ceiling della cpu */
__get_cpu_var(mrsp_state).cpu_ceiling = LITMUS_LOWEST_PRIORITY;

return err;
```

```
}
```

5.7 Schedule

Lo scheduler e' stato modificato quel tanto che basta per supportare il protocollo MrsP ma lasciando immutato il funzionamento di base di partitioned fixed priority.

Quanto finora discusso ha portato all'implementazione di due meccanismi che rispecchiano il modello ed il suo comportamente:

- se il job che viene prerilasciato detiene la risorsa non viene riaccodato nella coda ready del processore in cui si trova, viene invece aggiunto un flag ad esso per cercare in un secondo momento un processore in cui migrare
- se il job che sta per essere selezionato per eseguire, quindi si trova in testa alla coda ready, ha priorita' inferiore rispetto al ceiling il processore resta inutilizzato ed il job attende in coda.

```
static struct task_struct* pfp_schedule(struct task_struct * prev)
{
pfp_domain_t*   pfp = local_pfp;
struct task_struct*     next;

/* 1. determino lo stato attuale */
//  [...]
preempt = !blocks && (migrate || fp_preemption_needed(&pfp->ready_queue, prev));

/* 2. determino se il job attualmente in esecuzione e' il lock holder */
if(prev == pfp->sem->owner) {
        lock_holder = 1;
}

next = NULL;

/* 3. in base allo stato decido se devo scegliere un nuovo job da eseguire */
if ((!np || blocks) && (resched || !exists)) {

        /* 4. se il job prerilasciato e' il lock holder non lo riaccodo nella
        *  ready_queue della cpu corrente, altrimenti seguo la normale
        *  esecuzione di P-FP */
        if(prev && preempt && lock_holder) {
                /* Flag per la migrazione */
                tsk_rt(prev)->task_params.cpu = MIGRATION;
        } else {
                if (pfp->scheduled && !blocks && !migrate)
                        requeue(pfp->scheduled, pfp);
        }
```

```
        /* 5. se la risorsa e' occupata, controllo che il job in testa alla coda
        * ready abbia priorita' piu' alta del ceiling della cpu corrente.
        * Se il lock holder e' abbinato alla cpu corrente tale valore sara'
        * pari al ceiling della risorsa, altrimenti ha come valore
        * LITMUS_LOWEST_PRIORITY */

        if (pfp->sem->owner != NULL) {
                struct task_struct *t = fp_prio_peek(&pfp->ready_queue);

                if (t) {
                        if( get_priority (t) < __get_cpu_var(mrsp_state).cpu_ceiling +
                            1) {
                                placeholder = 0;
                        } else {
                                placeholder = 1;
                        }
                }

                /* 6. se il job in testa alla coda ready non ha priorita' piu' bassa
                      del
                *  ceiling la cpu restera' inutilizzata */
                if (placeholder == 0) {
                        next = fp_prio_take(&pfp->ready_queue);
                }
        }
} else
        /* 7. se esiste, continuo con il job corrente */
        if (exists)
                next = prev;

pfp->scheduled = next;

return next;
}
```

5.8 Context switch

Alla fine di ogni schedule viene richiamata la funzione pfp_finish_switch, in essa vengono gestiti due casi:

- viene riconosciuto il flag per la migrazione (punto 1) e viene gestita secondo le regole modello
- caso in cui si attua il meccanismo di notifica di MrsP, secondo al quale si forza ad eseguire il job che detiene la risorsa ed in quel determinato istante e' accodato nella coda ready di un processore.

```
static void pfp_finish_switch (struct task_struct *prev)
{
```

```
pfp_domain_t *to;

/* 1. flag per gestire la migrazione del lock holder */
if( get_partition(prev) == MIGRATION)) {

        /* 2. ripristino i parametri del job e cerco una cpu in cui migrare */
        tsk_rt(prev)->task_params.cpu = smp_processor_id();
        target_node = find_queue_entry(to->sem, smp_processor_id());

        if(target_node == NULL) {
                /* non vi sono cpu disponibili, quindi mi riaccodo nella cpu corrente
                   */
        } else {
                /* 3. modifico i parametri del job per la cpu di destinazione,
                *  il ceiling -1 in modo tale da poter prerilasciare il job
                *  che sta effettuando attesa attiva */
                tsk_rt(prev)->task_params.cpu =
                        (tsk_rt(target_node->task)->task_params.cpu);
                prev->rt_param.task_params.priority =
                        (to->sem->prio_ceiling[tsk_rt(prev)->task_params.cpu] - 1);
        }

        /* 4.  to ' indica la cpu corrente o la cpu target, nel primo caso il job
        *  non sara' piu' in esecuzione */
        to = task_pfp(prev);

        requeue(prev, to);
        if (fp_preemption_needed(&to->ready_queue, to->scheduled))
                preempt(to);

} else {

        /* 5. controllo se il lock holder e' accodato. */
        if(is_queued(local_pfp->sem->owner)) {

                target_node = find_queue_entry(to->sem);

                /* verifico se vi sono cpu disponibili per migrare */
                if(target_node != NULL) {

                        /* 6. migro nella cpu disponibile, facendo attenzione ad
                        *  aggiornare cpu e priorita' al ceiling corretto */
                        tsk_rt(local_pfp->sem->owner)->task_params.cpu =
                                (tsk_rt(target_node->task)->task_params.cpu);
                        local_pfp->sem->owner->rt_param.task_params.priority =
                                (to->sem->prio_ceiling[tsk_rt(local_pfp->sem->owner)->task_params.cpu]
                                - 1);

                        to = task_pfp(local_pfp->sem->owner);

                        fp_dequeue(local_pfp, local_pfp->sem->owner);

                        requeue(local_pfp->sem->owner, to);
```

```
                        if (fp_preemption_needed(&to->ready_queue,
                            to->scheduled))
                                preempt(to);
                        }
                }
        }
```