Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea Magistrale in Informatica

MULTIPROCESSOR RESOURCE SHARING PROTOCOL

IMPLEMENTAZIONE E VALUTAZIONE

Candidato:
Sebastiano Catellani

Relatore:
Prof. Tullio Vardanega

ANNO ACCADEMICO 2013 - 2014

Abstract

Multiprocess Resource Sharing Protocol propone un approccio innovativo per la condivisione di risorse globali. Burns e Wellings delineano una variante multiprocessor di *Priority Ceiling Protocol* con l'obiettivo di utilizzare, per sistemi multiprocessor partizionati, le tecniche di analisi di schedulabilità dei sistemi single processor. Per poterlo fare, occorre che il tempo di attesa per accedere alle risorse rifletta la contesa parallela, dovuta alla condivisione tra più processori, ma limitando il tempo di attesa dei job in coda senza precludere l'indipendenza dei job a priorità superiore che non la richiedono. MrsP prevede che l'esecuzione della sezione critica corrispondente alla risorsa, in caso di prerilascio del suo possessore, possa essere proseguita da parte del primo job della coda in attesa di accederla. In questa tesi, dopo aver analizzato nel dettaglio il protocollo, è elaborata una proposta di soluzione in grado di gestire anche gli aspetti che, nell'ambito teorico, non sono considerati. L'implementazione fornita è valutata tramite esperimenti che mirano a calcolare il costo delle singole primitive, valutare i costi aggiunti dal protocollo e, infine, confrontare MrsP con altri approcci.

Vorrei ringraziare il Prof. Tullio Vardanega, questa tesi non sarebbe stata possibile senza la sua guida e la sua immensa conoscenza, la quale è stata stimolo continuo sia in questa fase del mio percorso accademico sia negli anni precedenti.
Ringrazio mia mamma, per i mille sacrifici fatti in una vita difficile, con i quali è riusciata a dare a me, Paolo e Sara educazione e valori che ci hanno reso quello che siamo.
Senza il continuo sostegno della mia ragazza Valentina, neo esperta di sistemi real-time, difficilmente sarei arrivato dove sono, non solo all'università. La ringrazio per essermi accanto in ogni momento, felice o triste che sia.
Infine, voglio ringraziare i miei amici Seba, Fox, Ziby, Pigna e Dario, con cui ho vissuto giorno dopo giorno sin dal primo momento questa avventura, senza di loro sarebbero stati semplicemente anni di esami e studio, invece sono stati indimendicabili.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il progresso tecnologico ha portato a un rapido incremento della complessità del software e all'esigenza di prestazioni sempre maggiori da parte dell'hardware. Inizialmente, per aumentare la potenza di calcolo, si ricorreva a processori sempre più potenti, ma tale approccio causava problematiche significative come l'elevato consumo di energia e l'eccessiva dissipazione di calore. Per questo motivo i produttori, Intel in primis, hanno iniziato ad adottare piattaforme multiprocessor per lo sviluppo di sistemi real-time: affiancando più processori piuttosto che potenziarne uno unico.

Questa è la definizione che Burns e Wellings in [9] danno di un sistema real-time:

"An information processing system which has to respond to externally generated input stimuli within a finite and specified period. The correctness depends not only on the logical result but also on the time it was delivered. The failure to respond is as bad as the wrong response."

La ricerca si è quindi spostata sulle piattaforme multiprocessor. Nonostate i grandi sforzi e i recenti risultati, gli algoritmi di scheduling e le tecniche di analisi di schedulabilità per i sistemi multiprocessor non hanno ancora raggiunto il livello di maturità dei precedenti single processor (Davis et al. [10]).

Liu et al.[15] evidenziano come lo scheduling sia un problema intrinsecamente più complicato in ambito multiprocessor:

"Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling

problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors."

1.1 Contributo

Il lavoro di tesi è focalizzato sull'implementazione di Multiprocessor Resource Sharing Protocol a partire dalla versione di Partitioned-Fixed Priority fornita da LITMUS^{RT}. Quest'ultimo è un'estensione di Linux che ne permette l'utilizzo come sistema real-time. Esso mette a disposizione un sistema, basato su plugin, per l'implementazione di algoritmi di scheduling e protocolli per la condivisione di risorse. Il protocollo sviluppato, che si appoggia su tale sistema, è successivamente valutato empiricamente in termini di schedulabilità e di costi che l'implementazione stessa e i meccanismi utilizzati causano all'esecuzione.

1.2 Struttura

Nel primo capitolo 1 sono fornite le informazioni basilari riguardanti un sistema real-time, il suo modello e le categorie di scheduler; in seguito, è proposta una panoramica dei protocolli di accesso a risorsa single processor e multiprocessor, fino ad arrivare all'analisi di MrsP. Inoltre, capitolo presenta una panoramica su LITMUS^{RT}, che identifica il sistema di supporto per implementazione ed esperimenti. Nel capitolo 2, è discussa la soluzione alla base della tesi e la relativa implementazione; la sezione 2.3.7 descrive un esempio di esecuzione con l'obiettivo di dare risalto ai meccanismi del protocollo e in che modo interagiscono con le strutture dati. Gli esperimenti, atti a valutare il protocollo da diversi punti di vista, sono esposti e discussi nel capitolo 3. Infine, nel capitolo 4, sono tratte le conclusioni riguardo il lavoro di tesi e i possibili sviluppi futuri a partire da esso.

1.3 Introduzione ai Sistemi Real-Time

Questa sezione descrive il modello con task sporadici utilizzato e che trae origine dal lavoro di Mok et al. [18]. La scelta nasce dal fatto che il sistema LITMUS^{RT} è sviluppato a partire da questo modello, mentre MrsP, che comunque si basa su un modello sporadico, non ne specifica uno in particolare. Nel prosieguo del documento la nomenclatura originale subisce alcune modifiche coerentemente con il lavoro di Burns e Wellings [8].

1.3.1 Workload

Il *workload* consiste in un insieme di n task $\tau = \tau_1, ..., \tau_n$. Ogni task τ_i è ripetutamente invocato in modo asincrono da un evento esterno, per esempio un *interrupt* da parte di un dispositivo o lo scadere di un timer. Quando è invocato esso rilascia un *job* per gestire l'evento che l'ha generato. Il $j - esimo$ job di un task τ_i è identificato con $J_{i,j}$ ($j \geq 1$). Nei casi in cui l'indice del job risulti irrilevante J_i denota un qualsiasi job di τ_i.

Tasks. Ogni task τ_i è caratterizzato da una tripla di valori:

- C_i, *worst case execution time* (WCET), il tempo massimo richiesto per l'esecuzione;
- T_i, il periodo, il tempo minimo tra un evento di rilascio di un job ed il successivo;
- D_i, la deadline, lasso di tempo a disposizione per l'esecuzione.

Mentre il periodo e la deadline sono arbitrarie, il WCET dipende dalla piattaforma di esecuzione e dalla semantica del job. Nel primo caso dipende dalle componenti dell'hardware, per esempio la cache, la quale ha comportamenti che dipendono dalle esecuzioni precedenti che ne cambiano lo stato, o la frequenza di clock dei processori. Il secondo dipende dall'esecuzione del job, cioè dalla presenza o meno di istruzioni di *branch* che modifica il flusso delle operazioni variando il tempo di esecuzione. Il valore di WCET deve essere un limite massimo certo che rende il modello deterministico.

I valori che compongono la tripla sono soggetti ad alcuni vincoli:

- $C_i > 0$, il tempo di esecuzione deve essere non nullo;
- $T_i \geq C_i$, il periodo deve essere maggiore o uguale al WCET;
- $D_i \geq C_i$, la deadline deve essere maggiore o uguale al WCET.

Jobs. Un job $J_{i,j}$ diviene disponibile per l'esecuzione nel momento del rilascio $a_{i,j}$, con $a_{i,j} \geq 0$. La frequenza dei rilasci di un job è determinata dal periodo del task corrispondente: $a_{i,j+1} \geq a_{i,j} + T_i$. Ogni job $J_{i,j}$ richiede al massimo C_i

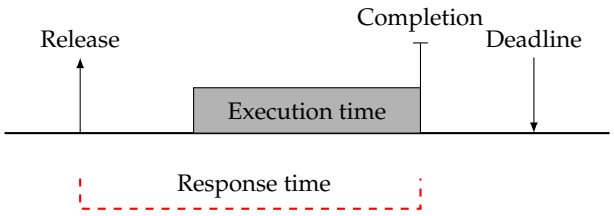


Figura 1.1: Job del modello.

unità di tempo del processore per completare la propria esecuzione entro $f_{i,j}$, con $f_{i,j} \geq a_{i,j}$. Un job si definisce *pending* dal momento di rilascio fino al suo completamento. Un task non può avere due job pending nello stesso momento; di conseguenza un job viene rilasciato solamente se il suo predecessore ha terminato l'esecuzione. Il *response time* di un job è pari al tempo in cui rimane pending; quindi $r_{i,j} = f_{i,j} - a_{i,j}$.
Il modello sporadico deriva dal modello a task periodico in cui viene rilassato il vincolo che obbliga un task ad avere rilasci strettamente periodici, cioè $ai, j = a_{i,1} + (j-1) * T_i$; tale limite diviene un valore minimo e non un vincolo stretto.

Deadline. La deadline relativa D_i determina la quantità di tempo a disposizione del job per il completamento. Il job $J_{i,j}$ deve eseguire entro la deadline assoluta $d_{i,j} = a_{i,j} + D_i$. Nel caso in cui l'esecuzione termini dopo la deadline assoluta, si è in presenza di *deadline miss* e si verifica un ritardo del job, formalmente definito *tardiness*. Una deadline miss ritarda il rilascio del successivo job. La relazione tra deadline e periodo permette di categorizzare i task:

- deadline implicite se $D_i = T_i$ per ogni $\tau_i \in \tau$;
- deadline vincolate se $D_i \leq T_i$ per ogni $\tau_i \in \tau$;
- deadline arbitrarie se non vi è alcun vincolo.

La categoria di deadline ha un'importante impatto sulle tecniche di analisi di schedulabilità mentre, dal punto di vista dell'implementazione, non comporta rilevanti differenze. Nel prosuguo di questo documento si assumeranno un insieme di task con deadline implicite.

Nella figura 1.1 si vede parte del modello discusso finora.

Processor demand. C_i indica il tempo di esecuzione richiesto da ogni job di τ_i, cioè per quanto tempo il job necessita di essere assegnato ad uno dei processori per eseguire il WCET entro la deadline. Data la presenza di una lunga serie di

job rilasciata da parte di un task, è utile normalizzare la domanda del processore mettendo in relazione periodo e deadline. Pertanto, si definisce un fattore di utilizzazione $U_i = C_i/P_i$ per ogni τ_i. Tale valore è importante in quanto identifica l'ammontare di tempo in esecuzione richiesto per l'intera durata di vita del task. Se tale necessità non è soddisfatta l'accumulo di tardiness può affliggere l'intero sistema.

Vincoli temporali. Un sistema è categorizzato in base ai vincoli temporali dei task che lo compongono, cioè in base a cosa consegue a una deadline miss:

- hard real-time, se causa un *fatal fault*;
- soft real-time, se è indesiderabile;
- firm, se non causa *fatal fault* ma l'utilità del risultato scende a zero.

1.3.2 Le risorse

Le risorse di un sistema si dividono in attive e passive. Un job per progredire nella propria esecuzioni ha bisogno della risorsa attiva, cioè del processore. Al contrario le risorse passive sono utilizzate dai job in quanto forniscono funzionalità, generalmente riutilizzabili a meno che il loro utilizzo non le esaurisca. Esse sono per esempio memoria, lock e mutex.

L'organizzazione dei processori permette di classificare i sistemi real-time in base al loro numero e al loro funzionamento. Sistemi con un unico processore sono definiti *single processor*; mentre *multiprocessor* indica sistemi con un numero di processori maggiore o uguale a due. In base alla loro configurazione, un sistema multiprocessor può essere:

- omogeneneo, se i processori che lo compongono sono identici per prestazioni e caratteristiche (cache, I/O bus, set di istruzioni, etc.): in questo caso il WCET di ogni job non dipende dall'unità su cui esegue e di conseguenza sono interscambiabili;
- uniforme, se i processori si differenziano per le prestazioni: i job eseguono con tempistiche differenti sui vari processori;
- eterogeneo, se i processori hanno differenti prestazioni e caratteristiche: non tutti i job possono eseguire su tutti i processori.

L'organizzazione dell'accesso alla memoria e le comunicazioni tra processori identifica due differenti categorie di sistemi: a memoria condivisa e a memoria distribuita. Nel primo caso i processori condividono un'unica memoria

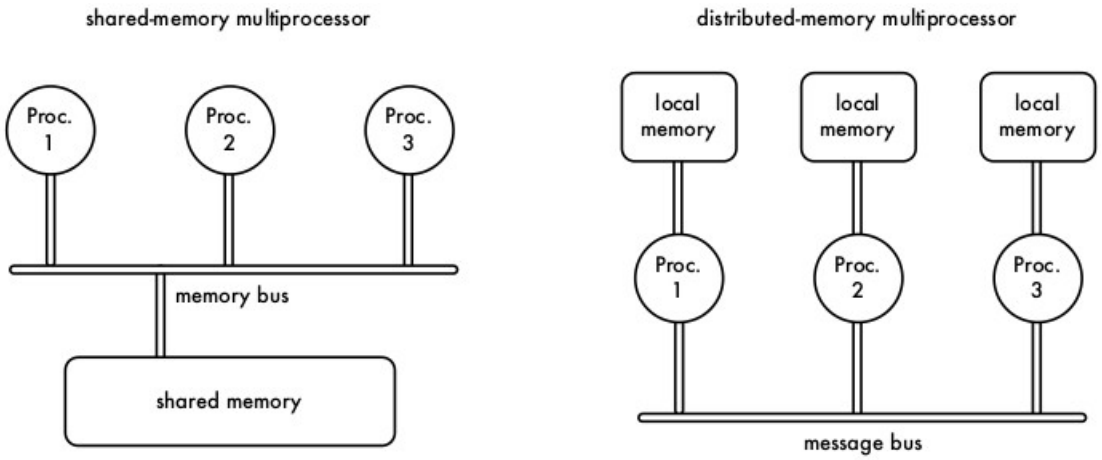


Figura 1.2: Illustrazione di architetture con memoria condivisa e distribuita.

centrale tramite un bus condiviso (architettura a sx in figura 1.2); nel secondo ogni processore (o sottoinsieme ristretto) ha una propria memoria e comunica tramite un bus dedicato allo scambio di messaggi (dx in figura 1.2). Il sistema a memoria distribuita va incontro ad alti overhead in caso di migrazioni, in quanto l'intero stato del processo migrante deve essere copiato da una memoria all'altra. Al contrario, il sistema a memoria condivisa necessita di copiare solamente i registri hardware da un processore all'altro, rendendo l'operazione meno onerosa.
Per questo motivo, il sistema a memoria condivisa è più utilizzato rispetto alla versione distribuita nonostante il bus condiviso per l'accesso alla memoria permetta l'accesso a un numero limitato di processo nel medesimo momento. Un sistema di questo tipo si differenzia a sua volta in due categorie in base alla gestione dell'accesso alla memoria: *uniform memory access* (UMA), se viene garantito uguale gestione a tutti processi, o *non-uniform memory access* (NUMA), nel caso contrario.

Data la sua semplicità, la maggior parte dei lavori in letteratura si fa riferimento a sistemi multiprocessor identici con memoria condivisa ad accesso uniforme, più comunemente definiti SMP (*symmetric multiprocessor platform*). Nel prosieguo del documento si adotterà tale architettura.

1.3.3 Scheduler

L'obiettivo di uno scheduler real-time è gestire l'esecuzione del sistema, adempiendo alle richieste dei task che compongono il workload entro i requisiti

temporali. Formalmente, un algoritmo di scheduling è un algoritmo che, dato una sequenza di job, deve determinare in ogni istante quale job deve eseguire e in quale processore. Il piano di esecuzioni risultante dall'algoritmo è definito *schedule*. Un algoritmo per essere valido deve rispettare i seguenti vincoli:

- in ogni istante, a ogni processore è assegnato al massimo un job;
- in ogni istante un job è assegnato al massimo a un processore;
- un job non è eseguito fino al momento del suo rilascio;
- la quantità di tempo del processore assegnata a ogni job è al massimo pari al suo WCET;
- un job esegue su un processore per volta.

Un algoritmo si definisce corretto se produce uno schedule valido. A sua volta uno schedule è definito *feasible* se ogni job esegue entro la propria deadline. Un taskset è *schedulable*, in relazione a un algoritmo di scheduling, se produce uno schedule feasible, di conseguenza è una proprietà che dipende dal taskset e non dall'algoritmo. Infine, un algoritmo è ottimo se genera sempre uno schedule feasible dato un qualsiasi taskset feasible.

Un test di schedulabilità determina se un algoritmo genera uno schedule feasible se applicato ad un particolare taskset e può essere sufficiente o necessario: nel primo caso un esito negativo indica un taskset non feasible, nel secondo caso un risultato positivo indica un taskset feasible. Un test contemporaneamente sufficiente e necessario è definito esatto.

1.4 Real-time scheduling in sistemi multiprocessor

Lo scopo dello scheduler è selezionare a ogni evento di scheduling un job dalla coda ready, cioè da quelli in attesa di eseguire. L'organizzazione e la gestione di tale coda permette di differenziare gli scheduler in base a diverse caratteristiche.

I sistemi in cui è presente un'unica coda sono definiti globali: in questo caso le esecuzioni su tutti i processori sono gestite prelevando i job da un'unica coda, e dunque essi possono eseguire su tutti i processori in modo indistinto. In un sistema partizionato, invece, i task sono suddivisi e allocati in un unico processore e, di conseguenza, vi è una coda per ogni processore sulla quale lo scheduler opera. Una versione ibrida tra i due sistemi prevede che i processori

siano divisi in sottoinsiemi e per ognuno di essi vi sia un'unica coda: tale configurazione prende il nome di scheduler a *cluster*.

L'ordinamento della coda avviene in base alla priorietà e la modalità di assegnazione di tale valore ai job permette di distinguere i seguenti algoritmi:

- task a priorità fissa: a ogni task è assegnata una certa priorità che viene applicata a ogni job che rilascia;
- job a priorità fissa: i job del medesimo task possono avere priorità differente, ma ogni job ha un'unica priorità; un esempio è *Earliest Deadline First* (EDF)
- priorità dinamica: la priorità di un job può assumere differenti valori, per esempio *Least Laxity First* (LLF).

L'approccio partizionato è largamente utilizzato in quanto permette di analizzare ed eseguire ogni singola partizione come un sistema single processor, con i conseguenti vantaggi dati da uno studio di tecniche e algoritmi maturi. Questo approccio, però, ha lo svantaggio di dipendere dalla fase offline di allocazione dei task tra i processori. Tale problema è riportabile al ben più noto *bin-packing*. Esso è uno dei problemi classici dell'informatica ed è NP-hard (Garey et al. [13]). Di conseguenza l'intero sistema dipende dalla risoluzione di tale problema e ciò porta a una soluzione al di sotto delle effettive prestazioni della piattaforma utilizzata, perché non si raggiunge un fattore di utilizzazione vicino all'ottimo. Inoltre, la condivisione di risorse tra task allocati in differenti processori causa un negativo impatto sulla schedulabilità del taskset. Al contrario, un sistema globale garantisce un alto livello di utilizzazione, ma comporta alti costi per la gestione di un'unica coda.

Davis et al. [10] propongono una panoramica dei principali scheduler in sistemi partizionati, clustered e globali. Nel prosieguo del documento approfondiremo i sistemi partizionati con utilizzo di task a priorità fissa, quindi con scheduler *Partitioned Fixed Priority* (P-FP).

1.5 Real-Time Locking Protocols

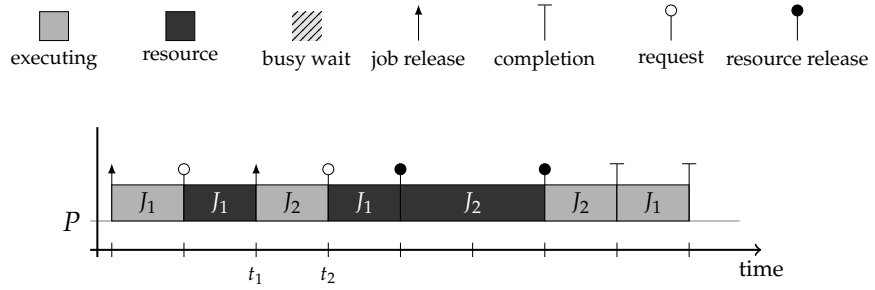
Il sistema descritto nella Sezione 1.3 assume che i task siano indipendenti, cioè che non condividano risorse diverse dal processore stesso. In un sistema a task indipendenti, gli m job ready con priorità maggiore (con m pari al numero di processori) eseguono. I job proseguono fino al loro completamento ogni volta che gli è assegnato un processore. Molti degli algoritmi di scheduling studiati e molte tecniche di analisi di schedulabilità sono basati su un sistema di questo

tipo. Tuttavia, in un sistema reale, i task condividono delle risorse. Si pensi, per esempio, a dispositivi di I/O, buffer, strutture dati, etc. I task non risultano più indipendenti e il progredire della loro esecuzione dipende dai job con cui condividono delle risorse.

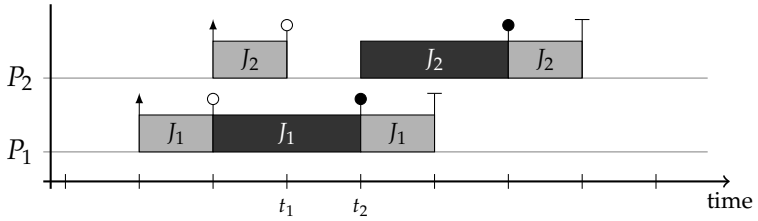
La condivisione di risorse necessita di meccanismi di sincronizzazione per prevenire situazioni di inconsistenza. L'utilizzo di *lock* permette di soddisfare questa necessità: il job che richiede una risorsa, se questa è in uso, attende il suo rilascio e, una volta libera, ne acquisisce il possesso. Potenzialmente, questo funzionamento preclude l'avanzamento dell'esecuzione nonostante sia assegnato a un processore. Sono possibili altri approcci, definiti *non-blocking*, che permettono al job di non attendere. Tali protocolli, però, sono molto onerosi per quantità di memoria richiesta o overhead generati.

In un sistema single processor, un job che richiede una risorsa occupata può solamente sospendersi in attesa del suo rilascio, per consentire al possessore di portare a termine la sezione critica (cioè la parte di esecuzione che richiede l'uso della risorsa e che necessita di sincronizzazione), altrimenti causerebbe stallo all'intero sistema (figura 1.3a). Le circostanze in cui un job a priorità inferiore esegue a discapito di uno a priorità superiore a causa della condivisione di risorse è chiamata *inversione di priorità*. Uno degli obiettivi principali di un protocollo di accesso a risorsa è limitare tale inversione perché rende complesso effettuare test di schedulabilità, oltre a snaturare il normale flusso dell'esecuzione in cui un job a priorità superiore esegue prima di uno a priorità inferiore. In un sistema multiprocessor, il job può sospendersi (figura 1.3b), come nel caso precendete, o effettuare l'attesa attiva fino al suo rilascio (figura 1.3c). L'utilizzo di attesa attiva ha lo svantaggio di sprecare l'esecuzione del processore ma ha il vantaggio di essere di semplice implementazione e causare un basso overhead; al contrario la sospensione, tra i vari svantaggi, causa l'allungamento del tempo di blocco subito da parte del job che richiede la risorsa. Per uno studio dettagliato della gestione del caso di blocco da parte di un job si veda Brandenburg et al. [5].

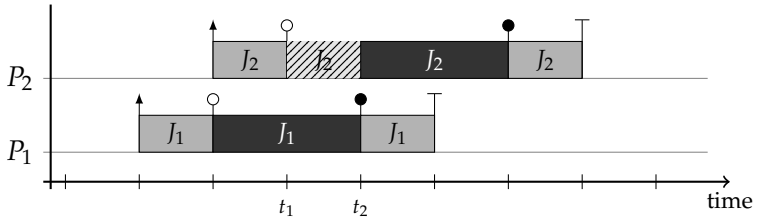
Un protocollo di accesso di risorsa deve garantire che i ritardi causati agli altri job siano limitati e calcolabili con precisione; allo stesso tempo, i job a priorità superiore non devono subire ritardi da job con cui non condividono risorse. Questo problema è stato risolto in ambito single processor con i protocolli PIP, PCP e SRP (Sezione 1.6). La maggior parte dei protocolli dei sistemi multiprocessor, invece, risolvono il primo problema, cioè limitano il ritardo per i job, ma non garatiscono l'indipendenza ai job a priorità superiore (Sezione 1.7).



(a) In ambito single processor, il job J_2 può solamente sospendersi al momento della richiesta della risorsa (t_2); altrimenti J_1, prerilasciato al tempo t_1, non rilascia la risorsa.



(b) In un sistema multiprocessor, il job J_2, al tempo t_1 può sospendersi in attesa che J_1 rilasci la risorsa (t_2).



(c) Con un approccio *spin-based*, J_2 effettua attesa attiva fino al rilascio della risorsa (t_1 - t_2).

Figura 1.3: Gestione delle risorse in ambito single processor e multiprocessor. Per ogni job $prio(J_i) = i$ e $i < j \implies prio(J_i) < prio(J_j)$.

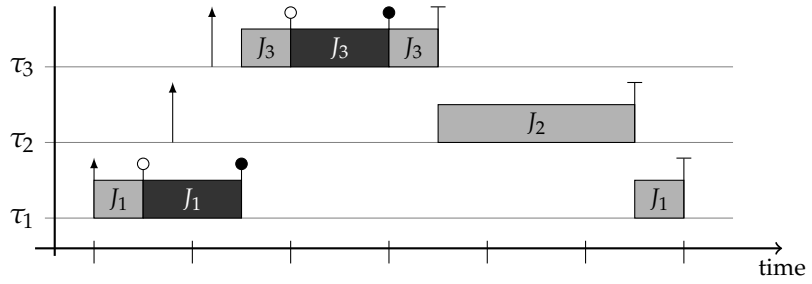


Figura 1.4: NPC - Gestione della risorsa tramite inibizione del prerilascio: J_1 non permette di eseguire ai job a priorità superiore fino a che non rilascia la risorsa.

Al contrario OMIP (Sezione 1.8) e MrsP (Sezione 1.9), con approcci differenti, garantiscono sia blocco ridotto che l'indipendenza dei job a priorità superiore.

1.6 Single processor Protocols

I protocolli per piattaforme con un unico processore sono stati ampiamenti studiati, in particolare sono presenti test di schedulabilità per FP e EDF che considerano l'inversione di priorità tra job data dalla condivisione di risorse.

Non-preemptive critical section protocol (NPC). Il modo più semplice per limitare il tempo di blocco causato dall'inversione di priorità è inibire il prerilascio durante la sezione critica. Pertanto, il job che richiede e ottiene la risorsa non viene prerilasciato dal job a priorità superiore fino a che non porta a termine l'esecuzione della risorsa. Questo funzionamento è illustrato in figura 1.4.NPC ha il vantaggio di essere facilmente implementabile e comporta bassi livelli di overhead, in particolare per task a livello kernel in quanto è sufficiente disabilitare gli *interrupt*.
Il blocco subito da un job a priorità superiore che richiede la risorsa occupata nel caso peggiore è pari alla lughezza della sezione critica stessa e avviene solamente una volta, precisamente prima dell'inizio della sua esecuzione. Ne consegue che NPC è facilmente integrabile nei test di schedulabilità. Lo svantaggio è che causa blocco anche ai job che non condividono la risorsa.

Priority inheritance protocol (PIP). Sha et al. [17] propone un protocollo che mira a non causare il blocco dei job con priorità superiore che non accedono alla risorsa. La particolarità del protocollo è che viene attivato solamente nei casi in cui il job che detiene la risorsa causi blocco a un job a priorità superiore.

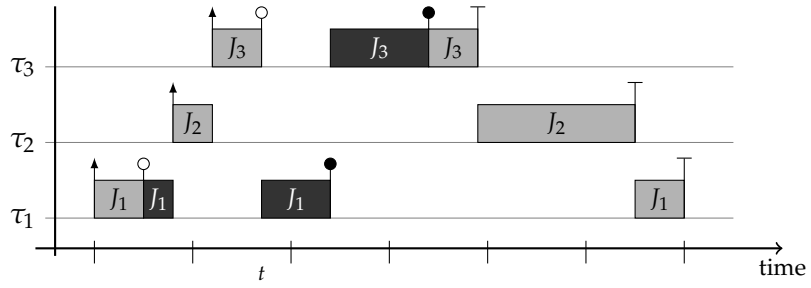


Figura 1.5: PIP - J_1 eredita la priorità di J_3 solamente al momento della sua richiesta di accesso alla risorsa (t).

In tal caso la priorità del job viene innalzata al valore massimo tra tutti i job in attesa in quel determinato istante (figura 1.5). Tale protocollo ha lo svantaggio che, in determinate circostanze, conduce a deadlock.

Priority-ceiling protocol (PCP). Sha et al. [17] presenta un protocollo disegnato principalmente per algoritmi FP e risolve il problema di deadlock del protocollo precedente. A ogni risorsa è abbinato un ceiling pari alla priorità massima tra tutti i job che durante l'esecuzione la richiedono; inoltre, è previsto un ceiling di sistema pari al ceiling più alto tra tutte le risorse in uso in un determinato momento. La richiesta di accesso da parte di un job è soddisfatta solamente se la sua priorità è superiore al ceiling di sistema. Una volta ottenuta la risorsa, se nesessario, il ceiling di sistema viene innalazato. Questo meccanismo di ceiling, illustrato in figura 1.6, garantisce che una richiesta di un job venga soddisfatta solamente se tutte le risorse di cui potrebbe necessitare sono libere.

Stack resource policy (SRP). Il protocollo delineato da Baker [2] è anch'esso basato su un sistema di ceiling. A ogni risorsa è abbinato un ceiling ed è presente un ceiling di sistema, entrambi calcolati e gestiti secondo le indicazioni di PCP. La differenza sostanziale sta nel fatto che a un job non è permesso di eseguire fino a che la sua priorità non è superiore al ceiling di sistema (figura 1.7). Ne consegue che un job esegue solamente se tutte le risorse di cui potrebbe necessitare sono libere. Pertanto il job subisce l'inversione di priorità solamente una volta e prima dell'inizio della sua esecuzione. SRP è alla base del protocollo studiato in questo elaborato per le sue proprietà fondamentali:

- un job è bloccato al massimo una volta durante la sua esecuzione;
- tale blocco avviene prima dell'inizio dell'effettiva esecuzione;

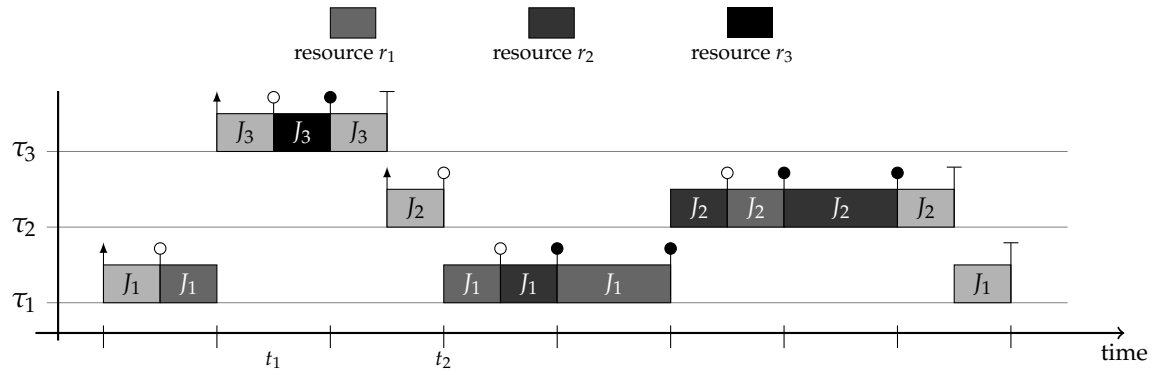


Figura 1.6: PCP - τ_1 e τ_2 condividono le risorse innestate r_1 e r_2; τ_3 accede alla risorsa r_3. La richiesta di accesso alla risorsa al tempo t_1 viene concessa in quanto j_3 ha priorità superiore rispetto al ceiling attuale, rispettivamente 3 e 2. Di conseguenza, il ceiling viene innalzato a 3. Al contrario, la richiesta di accesso al tempo t_2 viene ritardata in quanto il job ha priorità pari al ceiling, pertanto, deve attendere il rilascio delle risorse da parte di J_1 e il conseguente abbassamento del ceiling.

- quando il job inizia ad eseguire, tutte le risorse di cui necessita sono logicamente libere;
- previene situazioni di deadlock.

1.7 Multiprocessor Locking Protocols

Nei sistemi multiprocessor le risorse sono distinte in due categore: locali e globali. Nel primo caso i task che la richiedono sono tutti allocati nel medesimo processore, quindi possono essere utilizzati protocolli di accesso tipici dei sistemi single processor. Nel secondo caso, la risorsa è richiesta da job che eseguono su differenti processori.
I primi protocolli per sistemi multiprocessor sono il frutto di un riadattamento di protocolli single processor. Se in presenza di un unico processore la serializzazione degli accessi è intrinseca al fatto che esista un unico luogo di esecuzione, con un maggior numero di processori gli accessi sono potenzialmente paralleli e, di conseguenza, necessitano di meccanismi che permettano ai vari riadattamenti di soddisfare questa necessità.

Un'altra problematica, derivante dal fatto che i job su processori differenti condividono un'unica risorsa, sta nel fatto che non è significativo comparare le loro proprietà: un job con la priorità più alta nel proprio processore potrebbe

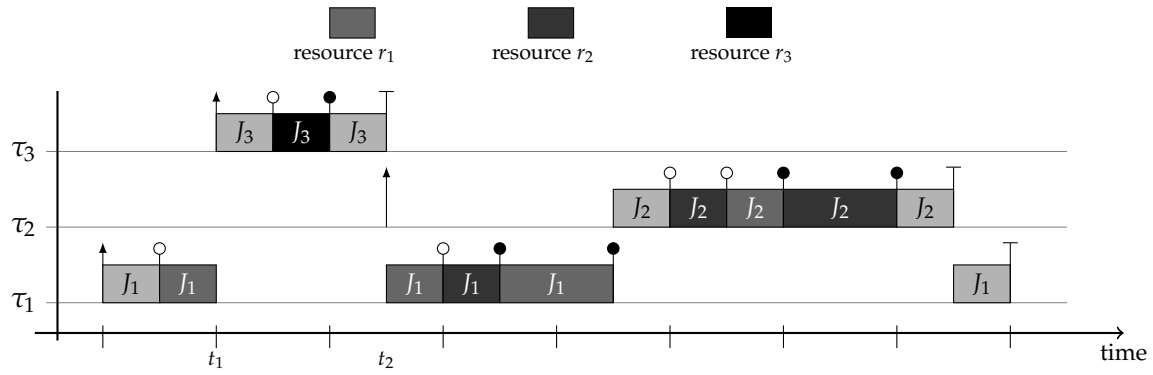


Figura 1.7: SRP - τ_1 e τ_2 condividono le risorse innestate r_1 e r_2; τ_3 accede alla risorsa r_3. Al tempo t_1, viene concesso di eseguire a j_3 in quanto ha priorità superiore rispetto al ceiling attuale, rispettivamente 3 e 2. Al contrario, l'esecuzione di j_2, rilasciato al tempo t_2, è ritardata in quanto ha priorità pari al ceiling, pertanto, deve attendere il rilascio delle risorse da parte di J_1 e il conseguente abbassamento del ceiling.

essere meno urgente rispetto a uno in un altro processore. I primi protocolli sviluppati per la condivisione di risorse globali ovviano a tale problema tramite il *boosting* del job, cioè innalzando la priorità del job al di sopra di tutte le priorità di base degli altri job. La differenza sostanziale tra la tecnica di boosting e l'inibizione del prerilascio sta nel fatto che tra *boosted job* avvengono prerilasci. Le richieste vengono poi gestite in base all'arrivo (FIFO) o tramite code ordinate secondo la priorità di base.

1.7.1 DPCP e MPCP

Le prime versioni derivano da un adattamento di PCP. Nonostante un approccio molto simile, i due protocolli si differenziano in quanto sono stati studiati per architetture differenti: in sistemi con memoria distribuita, ogni risorsa è accessibile solamente da un determinato processore nel quale è allocata; invece, in presenza di memoria condivisa, la risorsa è accessibile da ogni processore.

Distributed priority-ceiling protocol (DPCP). Rajkumar [16] utilizza RPC *remote procedure call* per gestire gli accessi alla risorsa: la richiesta viene presa in carico da un agente locale del processore della risorsa che esegue in modo sincrono e il job si sospende fino a che la richiesta non viene portata a termine. La priorità del richiedente resta invariata mentre viene aumentata quella dell'agente: assumendo che i sia l'indice del job richiedente ed n pari al numero totale di task nel sistema, la priorità dell'agente è innalzata a $n - 1$. L'indice è assegnato ai job in ordine di priorità effettiva; di conseguenza, i prerilasci tra

gli agenti rispecchiano le priorità dei task che li attivano. Gli agenti locali sono gestiti in base al protocollo PCP.

Con questo tipo di approccio, i job incorrono in diversi tipi di blocco e lo stesso accade per gli agenti nel processore di sincronizzazione. Pertanto, i test di schedulabilità sono particolarmente pessimistici.

Multiprocessor priority-ceiling protocol (MPCP). È un'evoluzione del precedente protocollo, progettato per sistemi con memoria condivisa. Le risorse globali possono essere accedute da ogni processore senza l'utilizzo di agenti. Una volta ottenuto l'accesso ad una risorsa, il job innalza la propria priorità a quella più alta tra tutti i task che la richiedono. Questo tipo di *priority boosting* velocizza l'esecuzione della sezione critica diminuendo l'ammontare di inversione di priorità subita dai job che condividono la risorsa o a priorità inferiore al ceiling, senza tuttavia creare blocco ai job con priorità superiore al ceiling globale. Dato che le priorità non sono uniche, a parità di valore non viene permesso il prerilascio, in questo modo non si ritarda l'esecuzione della sezione critica.
Anche in questo caso, i job che richiedono una risorsa occupata incorrono in diversi tipi di blocco in quanto l'accesso è garantito in base alla priorità, permettendo quindi ad altri job di accedere prima anche se l'hanno richiesta successivamente. Inoltre il blocco generato da altri job che non richiedono la risorsa potenzialmente possono ritardare l'esecuzione da parte del job in testa alla coda, causando così ulteriori ritardi al job stesso e agli altri in attesa.

MPCP, come DPCP, soffre di ritardi aggiuntivi derivanti dall'auto sospensione da parte dei job in attesa della risorsa. Lakshmanan et al. [14] propone una versione di MPCP che prevede virtual spinning, da cui MPCP-VS. Il protocollo rivisitato dispone che il job si auto sospenda, ma che nessun altro job del medesimo processore con priorità inferiore possa accedere ad una risorsa globale.

1.7.2 MSRP

Gai [12] propone un riadattamento del protocollo single processor SRP, da cui *Multiprocessor SRP*, anche se quest'ultimo non è utilizzato per le risorse globali, bensì per quelle locali. La gestione dell'accesso alle risorse globali è gestito tramite inibizione del prerilascio: quando un job effettua la richiesta inibisce il prerilascio e, se la risorsa è occupata, si accoda nella FIFO della risorsa corrispondente ed effettua attesa attiva sino al raggiungimento della testa;

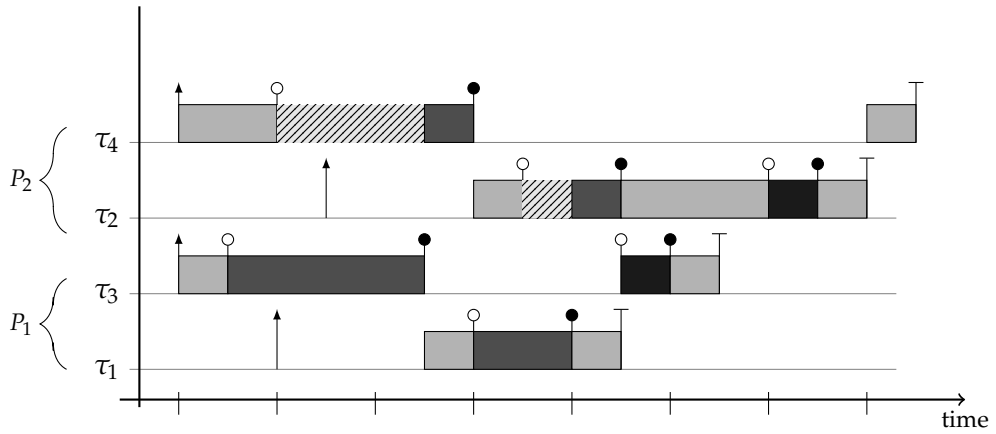


Figura 1.8: MSRP - τ_1, τ_2, τ_3 e τ_4 assegnati a 2 processori e 2 risorse.

una volta acquisita, esegue la sezione critica senza permettere ai job a priorità superiore del medesimo processore di eseguire (figura 1.8). Nella Sezione 1.6 si è visto come un approccio di questo tipo abbia vantaggi e svantaggi: l'attesa attiva comporta spreco di esecuzione del processore in cui il job la sta effettuando e causa blocco ai job a priorità più alta che condividono il processore ma non la risorsa, ma, allo stesso tempo, limita tale tempo di blocco, nel caso peggiore, alla lunghezza di un'unica sezione critica. Inoltre, dato che solamente un job alla volta per ogni processore può effettare una richiesta, la lunghezza della coda FIFO è al massimo pari al numero di processori. Di conseguenza, l'attesa da parte dei job in coda per la risorsa è limitata al tempo necessario per smaltire le richieste che lo precedono nella FIFO, quindi è determinata dalla sua lunghezza. I test di schedulabilità sono costruiti tenendo in considerazione questi tempi, i quali sono limitati grazie al protocollo di accesso a risorsa.

1.7.3 FMLP

Block in [3] propone un protocollo (*flexible multiprocessor locking protocol*) combinando i due differenti approcci suspensione-based e spin-based: le risorse sono suddivise tra brevi e lunghe in relazione alla durata della sezione critica. Le prime sono gestite tramite accodamento FIFO e inibizione di prerilascio mentre, nel secondo caso, tramite un protocollo che prevede sospensione. Con le risorse brevi è opportuno utilizzare inibizione del prerilascio, il quale è implementato in modo efficiente a livello kernel. Infatti, l'utilizzo di altri meccanismi, proprio per la durata della sezione critica, comportorebbe un costo superiore rispetto al beneficio. Per le risorse lunghe, invece, i job in attesa del rilascio della risorsa si sospendono in modo tale da non creare ritardi agli altri job, in particolare a

quelli a priorità superiore che non la richiedono. Oltre ad essere integrabile sia con scheduler globale che partizionato, questo protocollo permette accumulo di risorse tramite l'utilizzo di Group locks: risorse innestate che creano un unico gruppo al quale i job accedono in mutua esclusione; in tal modo, una volta ottenuto il lock sull'intero gruppo, è garantito che le risorse di cui necessita il job siano libere. Quest'ultimo approccio ha però il difetto di penalizzare il parallelismo, in quanto un job non permette ad altri job di accedere alle risorse che potenzialmente non utilizza, ma che fanno parte del medesimo gruppo.

1.7.4 Helping protocol

Con *Helping protocol* si identifica quell'insieme di protocolli in cui un job può prendersi carico dell'esecuzione della sezione critica, corrispondente ad una risorsa, per conto di un altro job. Nei protocolli discussi nei paragrafi precedenti è emerso che l'aspetto cruciale è la gestione dell'esecuzione della sezione critica: inibendo il prerilascio si danneggiano i job a priorità superiore che non la richiedono, ma, in questo modo, il tempo di blocco è limitato al tempo di utilizzo della risorsa. Al contrario, un innalzamento di priorità ad un valore precalcolato comporta la possibilità di essere prerilasciati, facendo aumentare il blocco subito dagli altri job o in attesa del risorsa o a priorità inferiore al ceiling del processore in questione. Questa categoria di protocolli prevede che l'avanzamento della sezione critica possa essere presa in carico da parte di uno dei job accodati qualora il suo possessore venga prerilasciato. Pertanto, l'esecuzione della risorsa viene portata a termine da un altro job ed il relativo risultato viene messo a disposizione del possessore al momento del suo risveglio. Attraverso questo meccanismo, il tempo di blocco ha durata massima pari alla lunghezza della sezione critica senza intaccare l'esecuzione dei job a priorità superiore.

SPEEP. (*Spinning Processor Executes for Pre-empted Processor* di Takada et al. [19]) si basa sull'assunzione che l'operazione corrispondente alla risorsa sia atomica: i job che ne necessitano accodano la propria richiesta nella corrispondente FIFO ed essa viene presa in carica dal primo job in coda in esecuzione. Il limite di tale protocollo è l'assunzione di partenza, cioè il fatto che la sezione critica sia atomica e quindi facilmente eseguibile da ogni job in coda.

M-BWI. Faggioli at el. [11] propongono *Multiprocessor Bandwidth Inheritance Protocol*, che rappresenta un approccio differente e maggiormente complesso.

In questo protocollo, i task eseguono all'interno di server fino all'esaurimento del suo *budget*. Il budget è la quantità di tempo, rinnovata ad ogni periodo, riservata ad un server per eseguire il job a cui è stato assegnato. I job che richiedono la risorsa effettuano busy-wait, senza inibire il prerilascio, fino a che non ne ottengono l'accesso, il quale è garantito in ordinamento FIFO. I job che attendono il rilascio utilizzano il budget del proprio server per effettuare attesa attiva oppure lo possono cedere al possessore della risorsa nel caso in cui esso venga prerilasciato. Ne consegue che, al contrario di SPEEP, il job non prenda in carico l'esecuzione per conto del job prerilasciato, bensì gli ceda l'esecuzione nel proprio processore per una quantità di tempo pari al budget residuo.

1.8 $O(m)$ Independence-preserving Protocol

OMIP è un protocollo studiato per sistemi con algoritmi clustered ed il suo obiettivo è limitare il tempo di blocco subito dai job in attesa della risorsa e da quelli che ne subiscono interferenza. Brandeburg in [4] dimostra come non sia possibile ottenere un limitato tempo di blocco e preservare l'indipendenza dei job a priorità superiore senza permettere migrazioni tra cluster. In sintesi, Brandeburg si sofferma sul concetto di *independence-preserving*, secondo il quale i job che non accedono alla risorsa non ne subiscono blocco. Infatti, un job subisce blocco per un tempo pari alla lunghezza della sezione critica solo se la richiede, altrimenti non subisce ritardi.

Il protocollo utilizza un approccio suspension-based, quindi un job che non riesce ad ottenere la risorsa si accoda e lascia l'esecuzione nel processore ai job a priorità inferiore. Indichiamo con W_i l'insieme di job in attesa che J_i rilasci la risorsa globale. Il meccanismo di *migratory priority inheritance* prevede che ogni qual volta J_i non sia in esecuzione, nonostante sia ready, ed esiste un job $J_x \in W_i$ tale che J_x è tra i c job a priorità più alta nel suo cluster, j_i migra nel cluster di J_x ereditandone la priorità e causando prerilascio nel cluster. Dopo il rilascio della risorsa, se necessario, J_i migra al proprio cluster.
L'intuizione alla base è quella di far migrare il job che detiene la risorsa a ogni prerilascio, scegliendo un cluster tra quelli in cui vi è un job in attesa e potenzialmente in esecuzione.

Il protocollo utilizza una serie di accodamenti:

- una coda ad ordinamento FIFO abbinata alla risorsa globale: essa ha lunghezza massima pari al numero di cluster del sistema e garantisce l'accesso alla

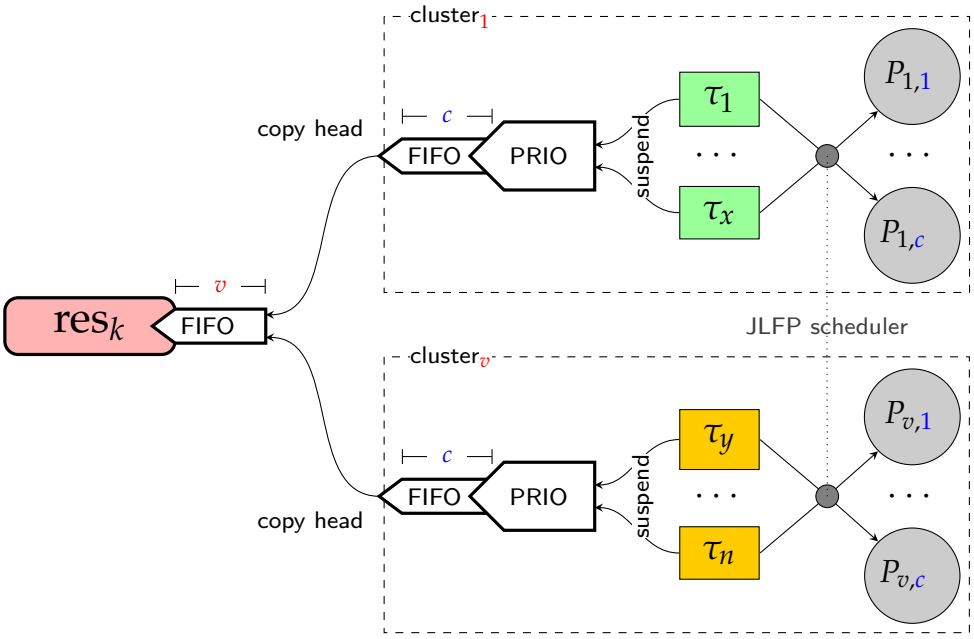


Figura 1.9: O(m) Independence-preserving Protocol.

risorsa al job in testa;

- una coda FIFO per ogni risorsa per ogni cluster: essa ha lunghezza limitata pari al numero di processori nel cluster e permette al job in testa di essere inserito in quella globale;
- una coda a priorità apposta a quella del punto precedente.

Il funzionamento alla base (figura 1.9) prevede che una richiesta venga accodata nella coda FIFO del proprio processore passando alla coda globale non appena raggiunge la testa. Nel caso in cui la prima coda sia piena, la richiesta viene inserita nella coda a priorità. Questo sistema ad accodamenti preserva l'indipendenza dei job e limita il tempo di blocco.

1.9 Multiprocessor Resource Sharing Protocol

Burns e A.J. Wellings in [8] descrivono le caratteristiche che un protocollo di accesso a risorsa per sistemi multiprocessor dovrebbe avere per gestire le criticità dovute alla sua condivisione viste in 1.5. Il loro obiettivo è la creazione di un protocollo che permetta l'utilizzo di tecniche di analisi di schedulabilità tipiche di sistemi single processor, nelle quali si tenga conto della serializzazione delle

richieste di accesso alla risorsa globale potenzialmente parallele.

Il modello adottato è quello approfondito in Sezione 1.3.1: n task (τ_i), caratterizzati da periodo T_i , deadline D_i e WCET, i quali generano una sequenza potenzialmente infinita di job. Ad ogni task è abbinato un valore di priorità $Pri(\tau_i)$.
La piattaforma di esecuzione consiste in m processori identici ($p_1 \dots p_m$).
Una risorsa r è condivisa da un insieme di task: essi devono accedere in mutua esclusione ed il codice corrispondente è definito *sezione critica*. Definiamo la funzione $G(r_j)$, la funzione che ritorna l'insieme di task che utilizzano la risorsa r_j, e $F(\tau_i)$, la funzione necessaria per ottenere l'insieme di risorse utilizzate da τ_i. Per semplicità di esposizione, si assume che il tempo di esecuzione di una risorsa sia identico per ogni task e lo si indica con c_j. Inoltre, definiamo map, la funzione che, dato un insieme di task, ritorna i processori in cui sono allocati.
Infine, indichiamo con e_j il parametro di tempo di esecuzione di r_j:

$$e_j = |map(G(r_j))| * c_j.$$

Dalla formula si evince che e_j indica il tempo massimo richiesto da un job per ottenere ed eseguire una risorsa, tempo che tiene conto dello smaltimento delle richieste già in coda e dell'esecuzione da parte del job stesso. Da ciò consegue la necessità che il protocollo permetta solo ad un job per processore di richiedere la risorsa. Questo evita che la coda superi la lunghezza di $|map(G(r_j))|$.

L'approccio di Burns e Wellings è un'estensione di PCP/SRP atta a gestire gli accessi alla risorsa in un sistema partizionato con scheduler P-FP. Ad ogni risorsa globale è abbinato un insieme di ceiling, uno per ogni processore: ogni valore è pari alla priorità massima tra i job richiedenti e allocati nello stesso processore.
MrsP eredita le caratteristiche di SRP (Sezione 1.6):

- un job è bloccato al massimo una volta durante la sua esecuzione;
- tale blocco avviene prima dell'inizio dell'effettiva esecuzione;
- quando il job inizia ad eseguire, non vi è nessuna richiesta pendendte alla risorsa dallo stesso processore;
- il protocollo impedisce deadlock.

MrsP ha l'obiettivo di riutilizzare la tecnica di analisi di schedulabilità single processor *Response-Time Analysis* (RTA) [1] che incorpora PCP/SRP:

$$R_i = C_i + max\{\hat{c}, \hat{b}\} + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \tag{1.1}$$

Il secondo addendo identifica il tempo di blocco subito dal job e causato dalla condivisione di risorse tra job a priorità inferiore con job a priorità superiore dal quale eredita la priorità. Quest'ultimo valore ($\hat{c}$) è messo a confronto con il costo introdotto dall'implementazione del protocollo stesso ($\hat{b}$). C_i rappresenta il WCET e le sezioni critiche corrispondenti alle risorse di cui necessita durante l'esecuzione (n_i indica il numero di volte che τ_i utilizza r_j):

$$C_i = WCET_i + \sum_{r^j \in F(\tau_i)} n_i c^j \tag{1.2}$$

Il passaggio di un protocollo di acceso a risorsa da un sistema single processor ad uno multiprocessor comporta un aumento del tempo necessario ai job per accedere alla risorsa e il conseguente ritardo causato ai job a priorità superiore che non la utilizzano. Di conseguenza, è necessario adoperare scelte algoritmiche che permettano di utilizzare le equazioni 1.1 e 1.2 aumentando il costo relativo all'accesso alle risorse, garantendo un tempo di attesa limitato per accedere alla risorsa e l'indipendenza dei job a priorità superiore al ceiling di ogni processore. Tale costo, relativo alla risorsa globale, non è pari ad una singola sezione critica c_j, come nel caso single processor, bensì incorpora la serializzazione degli accessi paralleli dati dalla presenza di più processori: e_j.

Le equazioni risultano quindi le seguenti:

$$R_i = C_i + max\{\hat{e}, \hat{b}\} + \sum_{\tau_j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \tag{1.3}$$

$$C_i = WCET_i + \sum_{r^j \in F(\tau_i)} n_i e^j \tag{1.4}$$

Un job, dopo aver inserito la richiesta di accesso nella FIFO della risorsa, innalza la propria priorità al valore di ceiling del processore in cui è allocato. Se è libera, ne acquisisce il possesso ed esegue la sezione critica. Al contrario, se la risorsa è occupata, il job effettua attesa attiva fintanto che la propria richiesta non

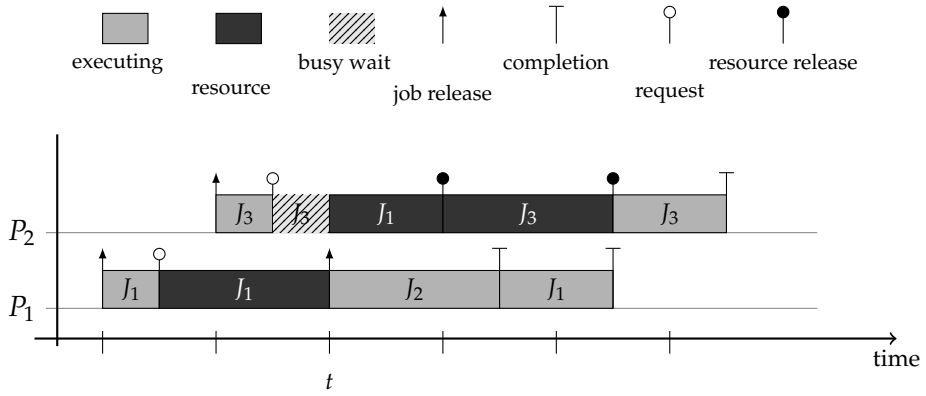


Figura 1.10: MrsP - Al prerilascio da parte di J_2 al tempo t, il job J_3, che sta eseguendo attesa attiva per ottenere l'accesso alla risorsa, prosegue l'esecuzione della sezione critica per conto di J_1.

raggiunge la testa della coda. Come visto in Sezione 1.5, se il job richiedente eseguisse con priorità più alta rispetto a tutti gli altri job con priorità superiore al ceiling, ottenendo quindi un comportamento pari all'inibizione del prerilscio, causerebbe blocco ai job che non richiedono la risorsa. Inoltre, se il job si sospendesse potrebbe subire ulteriori inversioni di priorità, aumentando così il tempo di blocco causato.

Il job che detiene la risorsa, nonostante esegua con priorità pari al valore di ceiling, limita il tempo di attesa dei job in coda (e quindi il tempo di blocco degli altri job) ma è ancora soggeto alla possibilità di prerilascio da parte dei job a priorità superiore. Il prerilascio causa l'aumento del tempo di attesa dei job in coda e, conseguentemente, il blocco subito dai job a priorità inferiore al ceiling. Per ovviare a tale problema, entrano in gioco meccanismi che permettano ai job in attesa di prendersi carico dell'esecuzione della sezione critica per conto del suo possessore (Sezione 1.7.4).

L'aspetto innovativo del protocollo MrsP sta nel prevedere che, in caso di prerilascio del possessore della risorsa, venga scorsa la coda delle richieste (in ordine di arrivo) e venga permesso al primo job che sta effettuando attesa attiva (quindi che non ha subito a sua volta prerilascio nel proprio processore) di eseguire la sezione critica per suo conto (figura 1.10).

Il funzionamento descritto è rappresentato graficamente in figura 1.11: in ogni processore le richieste sono gestite tramite un approccio simile al protocollo SRP che poi vengono accodate nella coda FIFO della risorsa ed effettuano attesa

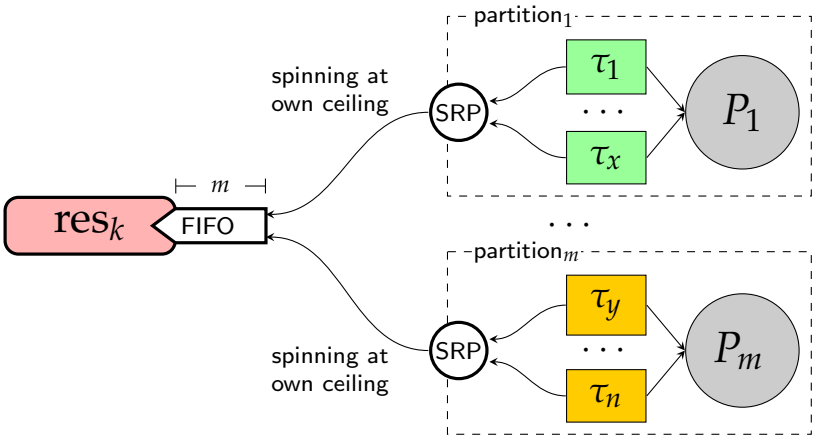


Figura 1.11: Multiprocessor Resource Sharing Protocol.

attiva fino all'acquisizione.

Le considerazioni fatte finora permettono di affermare che:

- al massimo un job per processore richiede la risorsa in un determinato istante;
- la lunghezza massima della coda FIFO di una risorsa r_k e' pari a $|map(G(r_k))|$;
- ogni job subisce blocco solamente una volta e prima della sua effettiva esecuzione;
- il tempo di attesa delle richieste in coda e il tempo di blocco subito dai job a priorità inferiore al ceiling è pari al massimo a e_j.

Le proprietà elencate rendono possibile utillizzare le equazioni 1.3 e 1.4 per verificare la schedulabilità di un taskset che utilizza MrsP come protocollo di accesso a risorsa.

1.10 Ambiente LITMUS^{RT}

LITMUS^{RT} (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) è un'estensione del kernel Linux, sviluppato dall'università della North Carolina a Chapel Hill, con lo scopo di creare e valutare algoritmi di scheduling e di accesso a risorsa per sitemi multiprocessor. Per facilitare la creziazione di plugin per l'implementazione di specifiche politiche di scheduling, LITMUS^{RT} fornisce un insieme di componenti (code, timer, ...), system call per i task real-time e una semplice interfaccia. Diversi plugin sono già sviluppati e messi a

disposizione da LITMUS^{RT} dalla release del 2013: EDF per sistemi partizionati, globali ed a cluster, partitioned fixed-priority ed algoritmi P-Fair. Inoltre, il sistema mette a disposizione un framework per il tracciamento di eventi, la fase di debugging e la valutazione delle primitive di scheduling e i costi del sistema.

1.10.1 Lo scheduler Linux

Il kernel Linux ha alcune limitazioni che non gli permettono di essere appropriato per applicazioni hard-real time, per esempio, è chiamato alla gestione di interrupt, potenzialmente di durata indeterminata, o di sezioni critiche non prerilasciabili. Ciò nonostante, esso è in grado di gestire un ampio sottoinsieme di applicazioni real time e permette di scalare senza particolari restrizioni su piattaforme multicore.

Lo scheduler di Linux è organizzato come una gerarchia di classi di scheduling. Ogni classe definisce una politica di scheduling e fornisce una serie di funzionalità: aggiungere un task alla classe, recuperare il prossimo task da eseguire, richiamare lo scheduler, e così via. Ogni classe è legata ad un'altra creando una lista. Ogni processo può essere eseguito in accordo con gli scheduler di differenti classi, ma deve appartenere ad una unica. Ogni qual volta sia necessaria una decisione di scheduling, viene percorsa la lista di classi sino a trovarne una con almeno un processo pendente. Di conseguenza, un processo di una classe a bassa priorità è eseguito solamente se i processi delle classi superiori sono inattivi.

Da un punto di vista implementativo, lo scheduler Linux è fondamentalmente partizionato, favorendo l'esecuzione di processi locali. Una coda di esecuzione è associata ad ogni processore e contiene lo stato di ogni classe di scheduling, il quale include, tra le altre cose, una coda dei processi ready, il tempo corrente e le statistiche di scheduling. Per lo più, la coda di esecuzione è protetta tramite lock contro gli accessi concorrenti. Dato che un processo appartiene a un solo processore, una migrazione richiede il lock di entrambe le code, di conseguenza, gli scheduler globali sono supportati con non poche difficoltà.

È importante osservere come i modelli teorici e i test di schedulabilità non tengano conto dei costi dovuti dallo scheduler stesso. Inoltre, gli eventi di scheduling non sono atomici e vi sono diverse situazioni in cui lo scheduler può cambiare di stato. Di conseguenza, tale problematica rende critica la

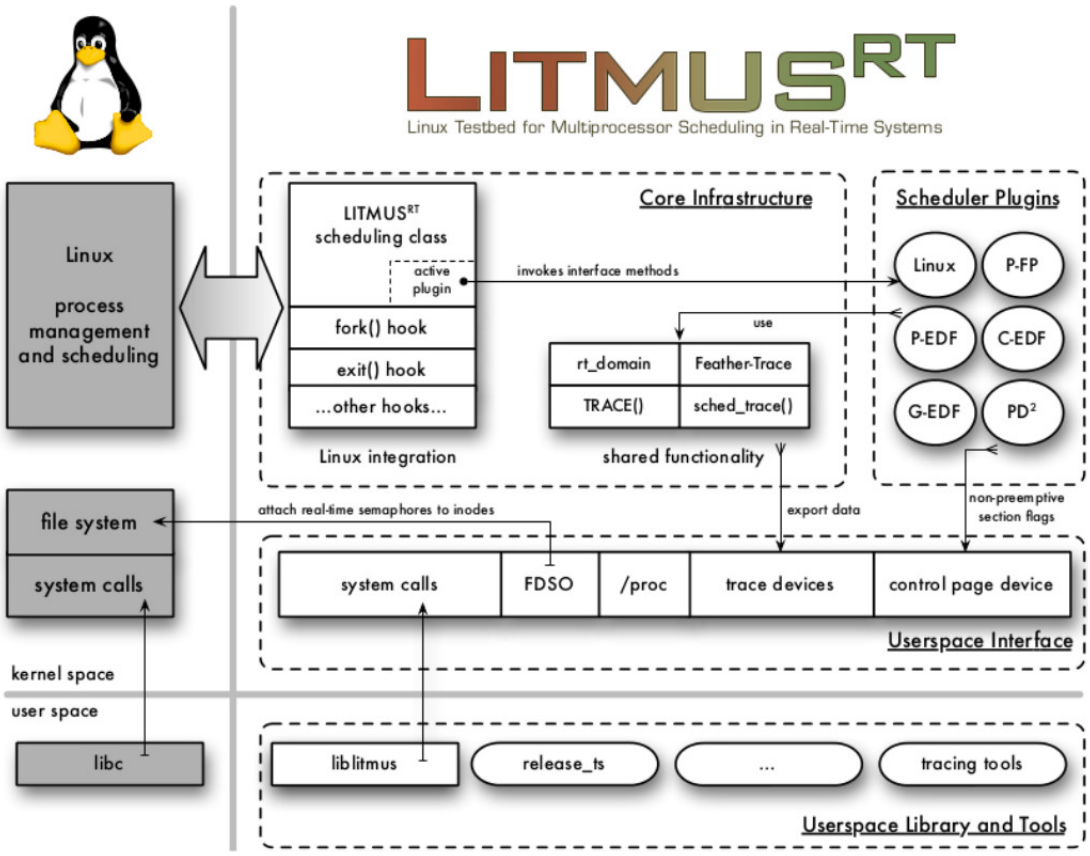


Figura 1.12: Illustrazione dell'architettura di LITMUS^{RT}.

gestione degli eventi in quanto si possono prendere decisioni erronee a causa di cambiamenti di stato durante la loro gestione.

1.10.2 Panoramica dell'architettura

L'infrastruttura di LITMUS^{RT} consiste di quattro componenti (figura 1.12):

- il core dell'infrastruttura;
- un numero di plugin, cioè gli scheduler a disposizione;
- un'interfaccia user-space;
- librerie e strumenti user-space.

Questa sezione descrive il core dell'infrastruttura e l'astrazione del dominio real-time ad alto livello. Per altri dettagli, come le interfacce del plug-in e delle

componenti user-space, si vedano l'appendice B e A.

Gli obiettivi del core sono (i) semplificare lo sviluppo e il mantenimento dei plugin e la loro correttezza, tutto ciò nascondendo dietro l'interfaccia la complessità del framework di scheduling di Linux, e (ii) fornire strutture dati e meccanismi facili da riutilizzare. Dato che LITMUS^{RT} è un'estensione di Linux, i task real-time sono implementati tramite dei processi standard: ogni ciclo di esecuzione del processo corrisponde a un rilascio di un job.
Riguardo al punto (i), LITMUS^{RT} aggiunge una nuova classe di scheduling in testa alla gerarchia, abbassando quindi di un livello la priorità della classe di Linux. Di conseguenza, le altre classi della gerarchia eseguono solamente quando non vi sono processi di LITMUS^{RT} in attesa. Un processo cambia di livello (da LITMUS^{RT} a Linux e viceversa) tramite system call.
La classe di scheduling di LITMUS^{RT} non implementa nessun algoritmo di scheduling, invece, astraendo tramite l'interfaccia del plugin, permette di integrare la logica di uno scheduler, cioè demanda le decisioni di scheduling e la gestione degli eventi al plugins. L'utilizzo di un layer intermedio come LITMUS^{RT} porta numerosi vantaggi, per esempio, l'astrazione dal sistema sottostante protegge dai cambi di versione del kernel Linux. Inoltre, fornisce un solido supporto per la migrazione per gli algoritmi globali ed a cluster. Riguardo a (ii), la sezione seguente discute l'interfaccia del plugin e il dominio real-time che fornisce code ready e code di release.

1.10.3 Astrazione del dominio real-time

Ogni scheduler richiede meccanismi che permettano di ordinare i job nella coda ready e accodarli per futuri rilasci. LITMUS^{RT} integra due strutture dati per soddisfare tali necessità: la coda ready e la coda di release. Quando un job viene rilasciato, viene trasferito dalla coda di release alla corrispondente coda ready, e lo scheduler viene invocato per controllora se è necessario un prerilascio. Tutti questi meccanismi sono forniti da una componente chiamata real-time domain (`rt_domain` nel codice). La sua implementazione utilizza il sistema di *hrtimers*, cioè timer hardware ad alta risoluzione, i quali sono fondamentali in quanto permettono di rispondere al rilascio di eventi e, quindi, di implementare algoritmi di scheduling basati su eventi.

Per una questione di ottimizzazione, la coda di release utilizza un timer per ogni tempo di release, piuttosto che un timer per ogni job. I job che condividono lo stesso tempo di rilascio sono organizzati in un albero binomiale, il quale permette di spostare i job dalla coda di release a quella ready in modo efficiente.

I timer sono associati ad ogni albero e programmati per scadere al momento del rilascio. I riferimenti ad ogni insieme di job che condividono la release sono memorizzati in una hash table per minimizzare il tempo di interazione. Infine, le due code sono protette da spinlock ([20]) che ne gestisce gli accessi e, inoltre, serializza le decisioni di scheduling.

La struttura rt_domain permette principalmente quattro operazioni:

- `add_release()` aggiunge un job al release heap
- `add_ready()` aggiunge un job alla coda ready
- `take_ready()` rimuove il job con la priorità più alta dalla coda ready.
- `peek_ready()` ritorna il job con la priorità più alta dalla coda ready senza rimuoverlo. Utilizzato per controllare se è necessario un prerilascio.

Per ordinare la coda ready deve essere definita un'operazione di confronto dato che la priorità di un job dipende dall'algoritmo di scheduling. Scheduler globali e partizionati si differenziano principalmente nell'uso della struttura rt_domain. Nel primo caso è definito un unico rt_domain ed è condiviso nello spazio globale. Al contrario, scheduler partizionati definiscono un rt_domain per ogni partizione (o cluster).

Capitolo 2

Implementazione proposta

La Sezione 1.9 discute il funzionamento di MrsP. Burns e Wellings in [8] riassumono le caratteristiche che l'algoritmo di scheduling deve avere:

- a ogni risorsa deve essere abbinato un insieme di ceiling, uno per ogni processore in cui è allocato almeno un job che la richiede;
- una richiesta di accesso a una risorsa deve innescare un innalzamento della priorità del job al *ceiling locale*, cioè il ceiling della risorsa corrispondente al processore in cui è stata effettuata;
- le richieste devono essere servite in ordine di arrivo;
- mentre il job attende di accedere alla risorsa e mentre esegue la sezione critica, deve eseguire con priorità pari al ceiling locale;
- ogni job che attende di accedere alla risorsa deve essere in grado di proseguire l'esecuzione della sezione critica per conto di ognuno degli altri job in attesa;
- il job incaricato per proseguire la sezione critica viene selezionato in ordine FIFO dalla coda delle richieste.

Inoltre, gli autori propongono due possibili soluzioni per realizzare il meccanismo tipico degli *helping protocol* (sezione 1.7.4), i protocolli che permettono a un job di riprendere l'esecuzione della sezione critica per conto del job che detiene la risorsa. Una soluzione deriva dal protocollo SPEEP, ma, come discusso in precedenza, è difficilmente applicabile perché presuppone che le azioni corrispondenti alla risorse siano atomiche. L'altra soluzione proposta, invece, prevede che un job in attesa ceda l'utilizzo del proprio processore al job che detiene la risorsa ed è stato prerilasciato, il quale potrà portare a termine la sezione critica.

Nella seconda soluzione, se un job viene prerilasciato da un job a priorità superiore mentre è in possesso di una risorsa, MrsP dispone che il lock holder possa migrare in uno dei processori in cui vi è un job in attesa attiva. Inoltre, il job acquisisce il valore di priorità direttamente superiore del ceiling del processore in cui migra, in tal modo ne causa il prerilascio ottenendo il possesso del processore. Questo meccanismo richiede che il valore di priorità indicato sia libero, cioè non assegnato a nessun task allocato nel processore. Una volta conclusa la sezione critica, il job, se necessario, migra al processore a cui è allocato.

Questo capitolo analizza l'implementazione proposta di MrsP: l'algoritmo alla base si ispira alla seconda soluzione proposta da Burns e Wellings. La sezione 2.1 fornisce un'analisi di dettaglio delle problematiche di implementazione concreta dell'algoritmo mentre, la sezione 2.2presenta le strutture dati necessarie per il suo funzionamento e, successivamente, espone in dettaglio l'implementazione dell'algoritmo.

2.1 Problematiche di progettazione

La progettazione deve mantenere le caratteristiche che rendono MrsP un protocollo innovativo, quindi (i) garantire un limitato tempo di attesa per accedere alla risorsa e (ii) preservare l'indipendenza dei job a priorità superiore. Come visto nella sezione 1.7, queste sono le problematiche principali che un protocollo di accesso a risorsa deve gestire in ambito multiprocessor. Un limitato tempo di attesa significa che un job che effettua una richiesta di accesso a una risorsa deve attendere, o effettuando attesa attiva o sospendendosi, per una quantità di tempo che sia limitata e che rifletta la contesa tra i job in esecuzione parallela in processori diffferenti. Con un approccio basato su attesa attiva, il tempo di attesa danneggia ogni processore in cui vi è un job interessato, in quanto quest'ultimo sta eseguendo a una priorità superiore rispetto a quella di base, ritardando, di conseguenza, l'esecuzione degli altri job.
Molti dei protocolli visti nel capitolo 1 soddisfano tale necessità tramite l'innalzamento della priorità del job al di sopra di tutti i job del processore oppure inibiendo il prerilascio. Questo fa si che essi adempiono al punto (i) a scapito del punto (ii), in quanto anche i job a priorità superiore che non richiedono la risorsa subiscono blocco.
Altri protocolli, al contrario, innalzano la priorità a un livello precalcolato senza interferire con i job a priorità superiore. Un esempio è MPCP (sezione 1.7.1), protocollo che però non soddisfa il punto (i): un job in coda per accedere alla risorsa deve attendere lo smaltimo delle richieste che lo precedono, ma se il job in possesso della risorsa viene prerilasciato da un job a priorità superiore

al ceiling, esso deve attendere, oltre alle singole sezioni critiche eseguite da parte di ogni job in coda, anche il loro completamento. Pertanto, per garantire indipendenza ai job più urgenti, il tempo di attesa e il blocco subito in ogni processore non è limitato.

Limitato tempo di attesa. Il punto (i) è soddisfatto tramite l'integrazione di diverse scelte implementative:

- usare a livello locale, cioè in ogni processore, *Immediate Priority Ceiling Protocol* (IPCP), il quale ha un comportamento simile a PCP/SRP in presenza di scheduler P-FP;
- porre il valore del ceiling locale pari alla priorità maggiore tra tutti i task che richiedono la risorsa nel processore;
- accodare la richiesta in una FIFO globale abbinata alla risorsa;
- se la risorsa è occupata, far effettuare attesa attiva al job che l'ha richiesta, cioè controllare ciclicamente la coda fintantoché la sua richiesta non ha raggiunto la testa della FIFO.

Queste operazioni garantiscono un limitato tempo di attesa per accedere alla risorsa in quanto: al più un job per processore contende per ottenere la risorsa in un determianto istante e le richieste sono servite in ordine di arrivo, evitando quindi situazioni di *starvation*. Di conseguenza, un job deve attendere, al massimo, il tempo di smaltimento delle richieste che sono state accodate prima della propria.

Independence-preserving. La modalità di calcolo del ceiling per ogni processore assicura l'indipendenza dei job a priorità superiore che non richiedono la risorsa (ii). Quest'ultimi, eseguendo, ritardano lo smaltimento della coda di richieste associata alla risorsa; pertanto, il job che detiene la risorsa deve migrare quando prerilasciato a condizione che vi sia un processore disponibile. Dato che non sempre è possibile migrare al momento del prerilascio, il meccanismo di base deve essere arricchito con un insieme di controlli che garantiscano che il job prosegua l'esecuzione ogni qual volta sia possibile. L'algoritmo prevede quindi di effettuare una migrazione, se necessaria, quando: o una nuova richiesta viene aggiunta alla coda o uno dei job in attesa torna a eseguire nel proprio processore. Inoltre, al momento del rilascio della risorsa, deve essere controllato lo stato del nuovo job in testa alla coda: se è stato prerilasciato, deve migrare in una delle CPU disponibili in modo da poter entrare in possesso della risorsa.

2.2 Strutture dati

La soluzione proposta fa uso di diverse strutture dati. Alcune messe a disposizione dall'implementazione dello scheduler P-FP e modificate per integrare MrsP, altre sviluppate appositamente per il protocollo. Di seguito, è fornita un'illustrazione delle principali strutture dati utilizzate, che rende più semplice l'esposizione e, di conseguenza, la comprensione delle sezioni di approfondimento dell'algoritmo implementato.

`pfp_domain`. È la struttura alla base dello scheduler P-FP. Incorpora la struttura `rt_domain` discussa nella sezione 1.10.3 e aggiunge le componenti necessarie per la realizzazione dello scheduling. In particolare, la coda dei job ready è implementata con una coda ordinata in base alla priorità (`fp_prio_queue`). Su di essa agiscono le primitive per la sua gestione (`fp_common.h`), mentre in riga 8 è definito uno spinlock, che previene stati inconsistenti e serializza le decisioni di scheduling. `pfp_domain` dispone di un puntatore alla struttura che gestisce la risorsa globale `mrsp_semaphore`, la quale è condivisa tra tutti i processori, e un puntatore a uno stato locale `mrsp_state`, instanziato per ogni processore.

```
1 typedef struct {
2     rt_domain_t                domain;
3     struct fp_prio_queue       ready_queue;
4     int                        cpu;
5     struct task_struct*        scheduled;
6     struct mrsp_semaphore*     sem;
7     struct mrsp_state*         mrsp_ceiling;
8 #define slock domain.ready_lock
9 } pfp_domain_t;
```

La struttura `pfp_domain`, essendo in presenza di uno scheduler partizionato, è istanziata per ogni processore tramite la macro `DEFINE_PER_CPU`. Durante il prosieguo del capitolo, in alcuni spezzoni di codice si useranno le variabili `local_domain` e `remote_domain`, essi fanno riferimento, rispettivamente, all'instanza locale del dominio di P-FP e a quella remota di un'altra partizione.

`mrsp_state`. Tiene traccia del livello di ceiling del processore in cui è istanziata. L'unica informazione che contiene è quindi un numero intero corrispondente al valore in vigore, il quale può essere pari o a `LITMUS_LOWEST_PRIORITY` (costante definita da LITMUS^{RT} che sta a inidicare il livello di priorità più basso ammesso nel sistema) o al ceiling della risorsa per quel determinato processore. `mrsp_state` è istanziato in ogni CPU tramite la macro `DEFINE_PER_CPU`. Esso

non prevede alcun spinlock per la gestione degli accessi in quanto, come si vede nella sezione 2.3.3, tale valore è modificabile solamente dopo aver ottenuto lo spinlock della risorsa.

```
1 struct mrsp_state {
2     int cpu_ceiling;
3 };
```

`mrsp_semaphore`. È la strutta dati che fornisce le informazioni utili per la gestione della risorsa globale secondo il protocollo MrsP:

- `litmus_lock`, struttura che permette di accedere all'interfaccia per la gestione delle risorse nell'ambiente LITMUS^{RT} (Appendice A);
- `lock`, spinlock che garantisce mutua esclusione nell'accesso alle informazioni della risorsa;
- `task_queue`, lista dei job che contendono per la risorsa; ogni nodo è un'istanza della struttura `queue_t` che contiene un puntatore al nodo successivo e uno al task (job) in attesa:

```
1 typedef struct queue_s {
2     struct list_head        next;
3     struct task_struct *    task;
4 } queue_t;
```

- `owner`, attuale possessore della risorsa, che, una volta acquisita, coincide con il task in testa alla `task_queue`;
- `prio_ceiling`, array che raccoglie i ceiling della risorsa, uno per ogni processore (`NR_CPUS`).

```
1 struct mrsp_semaphore {
2     struct litmus_lock    litmus_lock;
3     spinlock_t            lock;
4     struct list_head      task_queue;
5     struct task_struct    *owner;
6     int                   prio_ceiling [NR_CPUS];
7 };
```

L'istanza di `mrsp_semaphore` è condivisa tra i processori tramite la struttura `pfp_domain`.

`task_struct` e `rt_task`. Il task è l'entità di esecuzione principale nel sistema. Linux (linux/sched.h) fornisce una struttura di base che lo rappresenta e LITMUS^{RT} ne arricchisce l'espressività con delle componenti che ne permettono l'utilizzo in un sistema real-time. Alla sottostruttura `rt_task`, che contiene diverse informazioni tipiche di un task real-time, sono stati aggiunti i campi `home` e `priority_for_restore`: il primo indica la CPU in cui è allocato, utile in caso di migrazioni, mentre il secondo la priorità originaria, necessaria per il suo ripristino dopo il rilascio della risorsa.

`sched_plugin`. Infine, la struttura `sched_plugin` mette a disposizione l'interfaccia per la gestione degli eventi di scheduling. L'integrazione di MrsP necessita di operare al momento della decisione di schedule, scegliendo quale job far eseguire in un determinato istante, e dopo un `context switch`, cioè quando un job lascia l'esecuzione nel processore a un altro. Per maggiori informazioni riguardo l'interfaccia per lo sviluppo di un algoritmo di scheduling si veda l'appendice A.

Il sistema così configurato è illustrato in figura 2.1: a ogni processore sono abbinate informazioni riguardanti il dominio, necessarie allo scheduler, e allo stato locale della risorsa, quindi se è in uso e a che livello di ceiling. Inoltre, in ogni CPU è attivo il plugin per l'implementazione della logica dell'algoritmo di scheduling. I dati relativi alla risorsa globale, invece, sono unici e condivisi tra tutte le partizioni e il loro accesso è gestisto tramite l'interfaccia fornita da `litmus_lock`.

2.3 Algoritmo e implementazione

I paragrafi successivi descrivono le scelte algoritmiche effettuate in fase di progettazione con particolare attenzione all'implementazione. Una delle difficoltà incontrate durante lo sviluppo è stata quella di trovarsi in un ambiente a livello kernel. Si deve, quindi, tenere conto di diverse problematiche che nella teoria e nella progettazione ad alto livello non sono considerate. Tali informazioni sono integrate in questo capitolo in modo tale da giustificare alcuni campionamenti che vengono esposti nel capitolo 3, cioè operazioni che dal punto di vista algoritmico richiedono poche instruzioni, ma che, dovendosi calare negli interleaving di un kernel (esecuzioni in parallelo, interrupt, vincoli sui lock, ...), richiedono maggior attenzione.

Per ogni primitiva discussa si riporta il relativo codice in forma ridotta in modo tale da renderlo, con l'ausilio dei commenti, maggiormente leggibile.

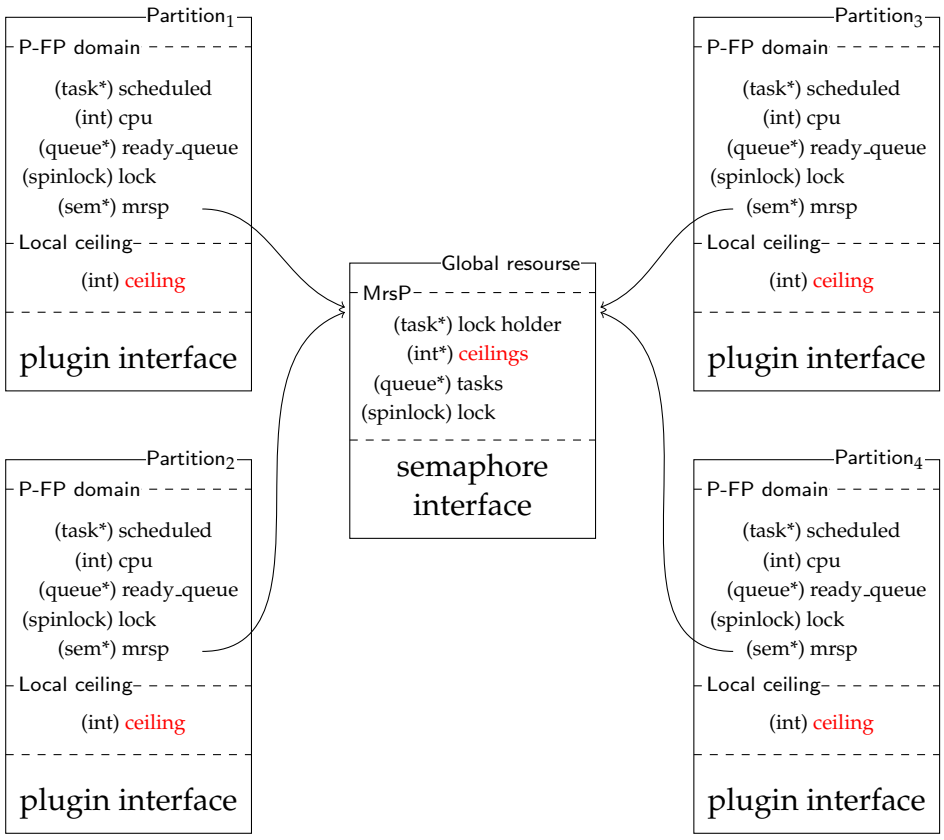


Figura 2.1: Organizzazione delle strutture dati in una piattaforma con 4 partizioni.

È messo in risalto l'uso degli spinlock, necessari per garantire mutua esclusione, e l'inibizione del prerilascio, per rendere minimo il tempo richiesto per effettuare un'operazione. Tuttavia, entrambi i meccanismi richiedono un attento utilizzo soggetto a diversi vincoli; per esempio, uno spinlock deve essere acquisito e rilasciato nella medesima primitiva; quando una primitiva è in possesso di uno spinlock non può effettuare operazioni bloccanti; un job può possedere solamente uno spinlock alla volta per evitare situazioni di deadlock. Il prerilascio, invece, è gestito tramite chiamate che vanno a operare su un contatore (`preempt_disable()` (+1) e `preempt_enable()` (-1)) e solamente a determinati valori possono essere effettuate alcune operazioni, come, per esempio, richiamare la funzione di scheduler.

2.3.1 Inizializzazione

In fase di inizializzazione, il protocollo richiede che siano predisposte le varie strutture dati necessarie al suo funzionamento: il plugin è registrato, i singoli domini in ogni processore e la struttura che rappresenta e gestisce la risorsa globale sono inizializzati. Inoltre, all'atto dell'accettazione dei nuovi task nel sistema, è calcolato il ceiling della risorsa per ogni processore aggiornando il valore di `mrsp_state`.

2.3.2 Gestione della coda

Le scelte algoritmiche sono incentrare sulla gestione della FIFO corrispondente alla risorsa in cui sono accodate le richieste di accesso. L'implementazione mira a gestire gli eventi che ne modificano lo stato, inteso come il numero di possibili processori in cui il job che detiene la risorsa può migrare. Perciò, risultano rilevanti le operazioni che aumentano tale numero: o l'inserimento di una nuova richiesta o una richiesta che ritorna attiva. In particolare, quando questo numero da zero, cioè il job possessore della risorsa non sta eseguendo, aumenta a uno, quindi vi e' un processore che può farlo proseguire. Per richiesta attiva si intende una richiesta il cui job corrispondente sta effettuando attesa attiva, inattiva in caso contrario.

Se la coda si trova nello stato descritto in figura 2.2, significa che il job, che detiene la risorsa, sta eseguendo nel proprio processore e che vi sono dei job in attesa. Di questi, alcuni stanno effettuando attesa attiva (contrassegnati dal colore verde) mentre altri sono stati prerilasciati da job a priorità superiore rispetto al ceiling del processore corrispondente (colore rosso). Per ogni richiesta, è necessario conoscere il job che l'ha effettuata, la partizione in cui è allocato e il relativo ceiling locale. In questa situazione (figura 2.2), il meccanismo di base di

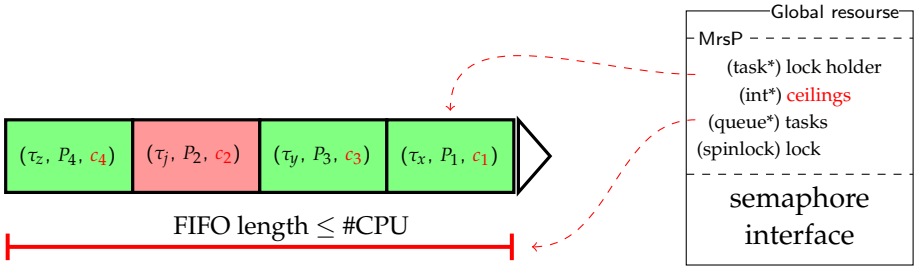


Figura 2.2: Coda delle richieste, esempio #1.

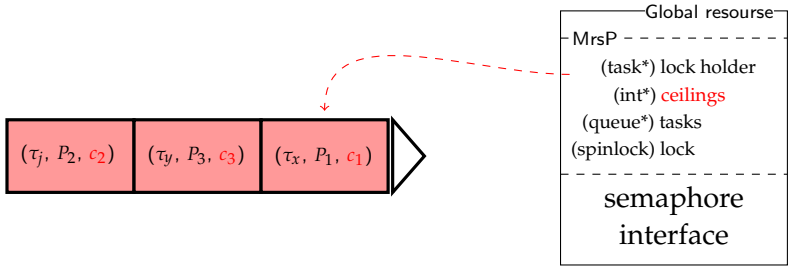


Figura 2.3: Coda delle richieste, esempio #2.

MrsP è sufficiente per garantire al lock holder di proseguire l'esecuzione in caso di prerilascio effettuando una migrazione nel primo processore disponibile in base all'ordine della coda (P_3 in questo esempio). L'implementazione di questo meccanismo è approfondito nelle sezioni 2.3.5 e 2.3.6.

Nel caso in cui non vi siano processori disponibili alla migrazione (figura 2.3), il job è inserito nella coda ready del processore in cui si trova, che potrebbe essere il processore in cui è allocato o uno in cui è migrato in precedenza. Anche se il job che detiene la risorsa non riesce a progredire nell'esecuzione della sezione critica, questo non comporta un allungamento del tempo di attesa delle richieste in coda e, conseguentemente, del tempo di blocco subito in ogni processore (sezione 1.9). Questo significa che in ogni processore, infatti, vi è un job a priorità superiore al ceiling che sta eseguendo e, tale ritardo, è inevitabile se si vuole garantire la loro indipendenza. Inoltre, tale *interferenza* è prevista nell'equazione 1.3.

In queste circostanze, l'algoritmo è chiamato a gestire quegli eventi in cui un processore o diviene o ritorna disponibile per la migrazione. Il primo caso (figura 2.4) avviene quando un job effettua la richiesta per accedere alla risorsa (sezione 2.3.3), mentre il secondo (figura 2.5) quando un job che stava effettuando attesa attiva ri-torna in esecuzione (sezione 2.3.6).

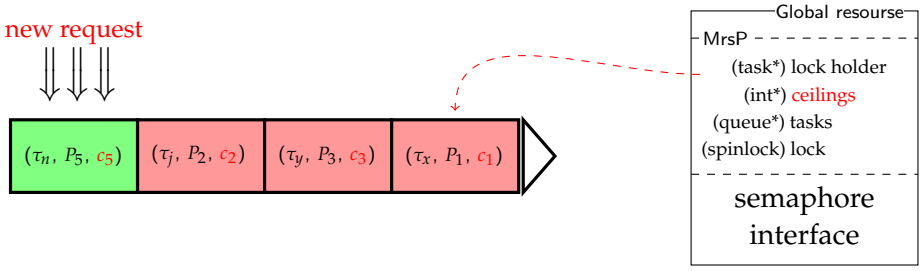


Figura 2.4: Coda delle richieste, esempio #3.

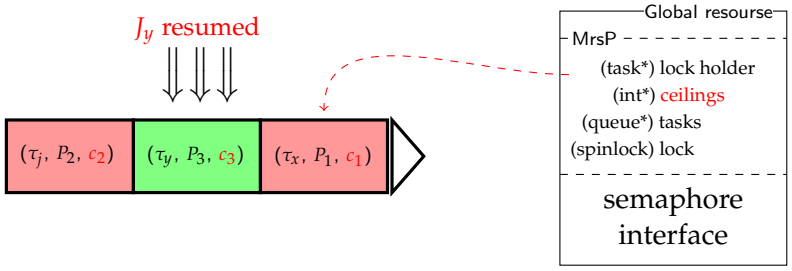


Figura 2.5: Coda delle richieste, esempio #4.

Infine, al momento del rilascio della risorsa, deve essere controllato lo stato del prossimo lock holder, cioè il job che ha raggiunto la testa della coda. Esso può trovarsi in due stati: in esecuzione, effettuando attesa attiva, oppure accodato nella `ready_queue` del proprio processore. Nel primo caso, il job entra in possesso della risorsa autonomamente (sezione 2.3.3). Se invece è stato prerilasciato, deve essere il protocollo a far migrare il job in modo tale da fargli acquisire la risorsa (sezione 2.3.4).

L'algoritmo così delineato soddisfa i requisiti di cui il protocollo MrsP necessita. Le prossime sezioni trattano le singole primitive in modo da dare risalto agli accorgimenti che rendono possibile il funzionamento dell'algoritmo nel suo complesso e la gestione degli interleaving e dei cambi di stato tipici di un software che opera in parallelo.

2.3.3 Richiesta di accesso

La primitiva `pfp_mrsp_lock` è eseguita in risposta alla richiesta di accesso effettuata da parte di un job. Essa può essere suddivisa in tre fasi definite: IPCP, migrazione e attesa attiva. La prima e la seconda fase operano sulla risorsa e sullo stato del processore corrente, di conseguenza, necessitano di essere ese-

guite con accesso esclusivo ai dati sensibili e senza subire prerilascio (righe 4 e 5): questo tipo approccio è ricorrente nell'implementazione in quanto assicura mutua esclusione nell'accesso ai dati, non permettendo quindi stati incosistenti, e non limita i ritardi rendendo le operazioni fondamentali non prerilasciabili.

L'implementazione di **IPCP** è compresa tra le righe 7 e 25: prevede l'innalzamento del ceiling locale e della priorità del job stesso e il controllo dello stato del possessore della risorsa. Il job che ha effettuato la richiesta eredita il valore di ceiling +1 se non ne ottiene subito il controllo, +2, invece, se la coda era vuota e quindi immediatamente disponibile. Questo accorgimento è necessario in quanto:

- il valore di ceiling è posseduto da uno dei task del processore (quello a priorità superiore tra tutti i task che la richiedono) e, in presenza di uno scheduler P-FP, potrebbe causare prerilascio [1];
- la priorità direttamente superiore al ceiling (+1) è assegnata ai job che effettuano attesa attiva;
- il valore di ceiling aumentato (+2) è assegnato al job che detiene la risorsa in modo tale da prerilasciare il i job del punto precedente.

Se necessario, il job cede l'esecuzione al possessore della risorsa tramite una **migrazione**. Questa operazione necessita dei lock dei due processori per trasferire il job da una coda ready all'altra. A livello di implementazione, questo comporta il rilascio del lock della risorsa (riga'27), l'acquisizione e il rilascio del lock del processore in cui è stato prerilasciato il job (righe 32 - 42) e di quello locale (righe 45 - 51). Il meccanismo appena descritto, serve a evitare che la primitiva accumuli più di un lock causando il deadlock dell'intero sistema. Infine, il protocollo forza un'operazione di schedule nel processore locale per permette l'effettiva migrazione (riga 56).

La terza e ultima fase dispone l'**attesa attiva** da parte del job. Essa prevede che il job ciclicamente acquisisca il lock della risorsa, controlli se la sua richiesta ha raggiunto la testa della coda (in tal caso acquiscisce la risorsa ed esce dal loop) e rilasci il lock.

```
1 void pfp_mrsp_lock() {
2     bool migration = false;
3
```

[1] L'implementazione di P-FP fornita da LITMUS^{RT} dispone che, a parità di priorità, vengano confrontati gli id dei due job.

```
4   preempt_disable();
5       spin_lock(mrsp->lock);
6
7           // Attivazione ceiling locale
8           mrsp_state.cpu_ceiling = (mrsp->ceilings[get_partition(task)]);
9           // Accodamento della la richiesta alla coda
10          task_queue_add(mrsp, task);
11          // Riferimento al task in testa alla coda
12          next = get_head(mrsp->task_queue);
13
14          // Risorsa libera e richiesta in testa?
15          if (sem->owner == NULL && next == t) {
16              // Il job entra in possesso della risorsa e innalza la priorita'
17              mrsp->owner = task;
18              task->priority = (mrsp->ceilings[get_partition(local_domain)] + 2);
19          } else {
20              // Il job innalza la priorita'
21              task->priority = (mrsp->ceilings[get_partition(local_domain)] + 1);
22              // L'attuale possessore della risorsa sta eseguendo?
23              if (is_queued(mrsp->owner))
24                  migration = true;
25          }
26
27      spin_unlock(mrsp->lock);
28
29      if (migration) {
30          bool still_queued = true;
31
32          spin_lock(remote_domain->lock);
33
34              // Il job e' ancora accodato?
35              if (is_queued(mrsp->owner)) {
36                  // Rimozione del job dalla ready_queue in cui e' accodato
37                  task.cpu = target_cpu;
38                  fp_dequeue(remote_domain, mrsp->owner);
39              } else {
40                  still_queued = false;
41              }
42          spin_unlock(remote_domain->lock);
43
44          if (still_queued) {
45              spin_lock(local_domain->lock);
46
47                  // Il job acquisisce il ceiling locale ed e' aggiunto alla ready_queue
48                  mrsp->owner->priority = (mrsp->ceilings[local_domain] + 2);
49                  requeue(mrsp->owner, local_domain);
50
51              spin_unlock(local_domain->lock);
52          }
```

```
53
54              if (still_queued) {
55                  // Il protocollo forza l'operazione di schedule() per cedere il processore
56                  schedule();
57              }
58          }
59
60      preempt_enable();
61
62      // Attesa attiva
63      if (mrsp->owner != task) {
64          do {
65              spin_lock(mrsp->lock);
66
67                  // Riferimento al task in testa alla coda
68                  next = get_head(mrsp->task_queue);
69                  // Risorsa libera e richiesta in testa?
70                  if (sem->owner == NULL && next == t) {
71                      // Il job entra in possesso della risorsa e innalza la priorita'
72                      mrsp->owner = task;
73                      task->priority = (mrsp->ceilings[task->cpu] + 2);
74                  }
75
76              spin_unlock(mrsp->lock);
77          } while(mrsp->owner != task);
78      }
79  }
```

2.3.4 Rilascio della risorsa

Il rilascio della risorsa richiede di modificare le informazioni contenute nella struttura dati che rappresenta la risorsa globale, di conseguenza richiede l'acquisizione del relativo spinlock. Inoltre, è necessario inibire il prerilascio per evitare che le operazioni vengano interrotte. Innanzitutto, l'implementazione attua il rilascio della risorsa e il ripristino della priorità del job e del ceiling. Da notare, riga 9, `mrsp_state` è un riferimento alla struttura istanziata nel processore di origine del job, mentre la primitiva potrebbe essere eseguita in un altro processore a conseguenza di una migrazione. Ripristinando il ceiling prima della migrazione di ritorno, si limita il tempo di blocco subito nel processore di origine.
Un volta rilasciata la risorsa, la primitiva controlla lo stato del prossimo possessore della risorsa, cioè quello corrispondente alla richiesta che ha raggiunto la testa della FIFO. Se è stato prerilasciato e si trova in una coda ready si prosegue con una migrazione, a meno che non sia il job che ha appena rilasciato la risorsa

ad averlo prerilasciato in precedenza dopo una migrazione (righe 15 - 33). La migrazione avviene dopo aver rilasciato lo spinlock della risorsa e necessita, come visto nella sezione precedente (2.3.3), degli spinlock di due partizioni, facendo attenzione ad aggiornare la sua priorità e il parametro che indica la CPU in cui si trova.
Infine, il job, se non è nel suo processore di origine (`task->home`), migra nella partizione in cui è allocato il corrispettivo task (righe 64 - 74).

```
1  void pfp_mrsp_unlock() {
2
3      preempt_disable();
4          spin_lock(mrsp->lock);
5
6              // Rilascio della risorsa
7              mrsp->owner = NULL;
8              // Ripristino del ceiling nella cpu di orgine (cosi' da limitare il tempo di blocco)
9              mrsp_state.cpu_ceiling = LITMUS_LOWEST_PRIORITY;
10             // Ripristino la priorita' del job che ha rilasciato la risorsa
11             task.priority = task.priority_for_restore;
12             // Rimozione della richiesta dalla testa della coda
13             task_queue_pop(mrsp);
14
15             // Lista delle richieste vuota?
16             if (!list_empty(mrsp->task_queue)) {
17                 // Riferimento al task in testa alla coda
18                 next_lock_holder = get_head(mrsp->task_queue);
19                 // E' accodato nella sua cpu?
20                 if (is_queued(next_lock_holder)) {
21                     // E' il job che sta rilasciando la risorsa a bloccarlo a causa di una migrazione?
22                     if (next_lock_holder->cpu != task->cpu) {
23                         // C'e' un processore disponibile?
24                         cpu = find_cpu_available(mrsp);
25                         // Set-up per la migrazione
26                         if (cpu != NULL) {
27                             next_owner = next_lock_holder;
28                             from_cpu = next_owner->cpu;
29                             target_cpu = cpu;
30                         }
31                     }
32                 }
33             }
34
35         spin_unlock(mrsp->lock);
36
37         // E' necessario far migrare il prossimo lock holder?
38         if (next_owner) {
39             bool still_queued = true;
```

```
40
41          spin_lock(from->lock);
42              // Il job e' ancora accodato?
43              if (is_queued(next_owner)) {
44                  // Rimozione del job dalla ready_queue in cui e' accodato
45                  next_owner.cpu = target_cpu;
46                  fp_dequeue(from, next_owner);
47              } else {
48                  still_queued = false;
49              }
50          spin_unlock(from->lock);
51
52          if (still_queued) {
53              spin_lock(target->lock);
54
55                  // Il job acquisisce il ceiling locale ed e' aggiunto alla ready_queue
56                  next_owner->priority = (mrsp->ceilings[target] + 2);
57                  requeue(next_owner, target);
58                  preempt(target);
59
60              spin_unlock(target->lock);
61          }
62      }
63
64      // E' necessario far migrare il job alla propria CPU di origine?
65      if (task->cpu != task->home) {
66          // Ripristino della CPU di origine
67          task->cpu = task->home;
68          // La migrazione avviene sfruttando il meccanismo messo a disposizione da LITMUS, il quale
                avviene dopo il context-switch
69          preempt(local_domain);
70      } else {
71          // Il job e' nella cpu di orgine, e' quello a priorita' piu' alta?
72          if (!highest_job(local_domain, task))
73              preempt(local_domain);
74      }
75
76  preempt_enable();
77 }
```

2.3.5 Scheduling

L'implementazione fornita da LITMUS^{RT} predispone una funzione di schedule che rispecchia la logica dello scheduler P-FP: dopo aver determinato lo stato corrente del processore e del job attualmente in esecuzione, la primitiva, se necessario, confronta il job in testa alla coda ready con quello attualmente in esecuzione e, se ha priorità superiore rispetto a quest'ultimo, effettua un

prerilascio causando un *context switch*. Di conseguenza, il job prelevato dalla coda inizia a eseguire mentre il job prerilasciato viene posizionato nella coda dei job ready in base alla sua priorità.

È stato necessario modificare la primitiva per garantire il supporto al protocollo MrsP. Le funzionalità introdotte sono due: la prima deriva dal protocollo in sé, mentre la seconda deriva da una scelta algoritmica che mira a ridurre il numero di migrazioni.

Lock holder prerilasciato. Nel caso in cui il job precedentemente in esecuzione (`prev`), quindi prerilasciato, sia in possesso della risorsa, il paramentro CPU del task viene utilizzato come flag per eseguire la migrazione a context switch avvenuto (riga 9 e 18). Come visto in sezione 2.3.3 e 2.3.4, la migrazione consiste nel trasferire il task nella coda ready del processore selezionato. Potrebbe risultare logico effettuare l'operazione in questa primitiva, ad esempio sostituendo in riga 18 `prev.cpu = MIGRATION` con `requeue(prev, remote_domain)`. Purtroppo, tale approccio non è possibile in quanto la primitiva `pfp_schedule` necessita dello spinlock locale (riga 25 e 45) per garantire l'integrità del dominio e serializzare le decisi di scheduling. Di conseguenza, non è possibile ottenere lo spinlock né della risorsa né del processore in cui far migrare il job.
Il flag utilizzato permette di posticipare la migrazione alla primitiva `pfp_finish_switch` (sezione 2.3.6), la quale, come vedremo, non è vincolata da alcun tipo di spinlock.

Placeholder. Burns e Wellings in [8] non considerano una particolare circostanza che si viene a creare quando un job migra in un processore diverso dal proprio. L'idea principale su cui costruito il protocollo consiste nel permettere solamente a un job per processore di effettuare una richiesta di accesso alla risorsa. Questo comportamento è garantito dal fatto che il job che richiede/detiene la risorsa effettua attesa attiva/esegue la sezione critica a un livello di priorità superiore rispetto agli altri job che la potrebbero richiedere e che, quindi, non eseguono fino a che non hanno ottenuto e rilasciato la risorsa.
Si supponga che il sistema si trovi nello stato seguente: al tempo t_i, J_1 acquisisce la risorsa mentre sta effettuando attesa attiva, esso si trova quindi nel processore (P_1) in cui il task padre τ_1 è allocato. Al tempo t_{i+1}, J_2 viene rilasciato nel medesimo processore e causa prerilascio a J_1 in quanto ha priorità superiore rispetto al ceiling. J_1 migra in P_2 prerilasciando J_3, il primo processore disponibile tra

quelli in coda. La situazione descritta non ha nulla di particolare e i meccanismi discussi sono sufficienti a soddisfare i requisiti del protocollo. Al tempo t_{i+2}, J_2 completa la propria esecuzione, mentre J_1 conclude la sezione critica al tempo t_{i+3} e migra al processore di origine.
Nel tempo compreso tra t_{i+2} e t_{i+3}, lo scheduler della partizione P_1 assegna l'esecuzione al job a priorità più alta, cioè in testa alla coda ready, il quale richiede l'accesso alla risorsa globale.

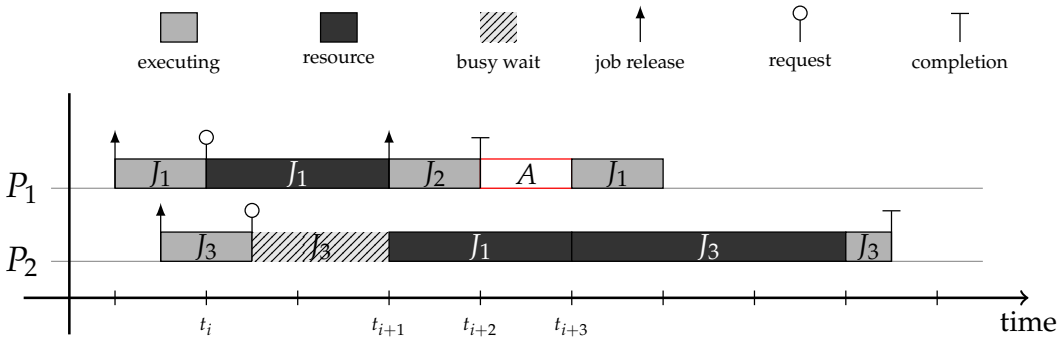
La situazione descritta evidenzia la necessità di un ulteriore meccanismo che venga attivato al momento del completamento di J_2. Di seguito sono descritte e illustrate (figura 2.6) delle possibili soluzioni:

a) in P_1 si attiva un agente che esegue con priorità pari al ceiling, simulando la presenza di J_1 nel processore;
b) si forza J_1 a migrare in P_1 e in P_2 ricomincia a eseguire J_3;
c) lo scheduler cede l'esecuzione in P_1 al job in testa alla coda ready solamente se ha priorità superiore al ceiling, in caso contrario il processore resta inutilizzato.

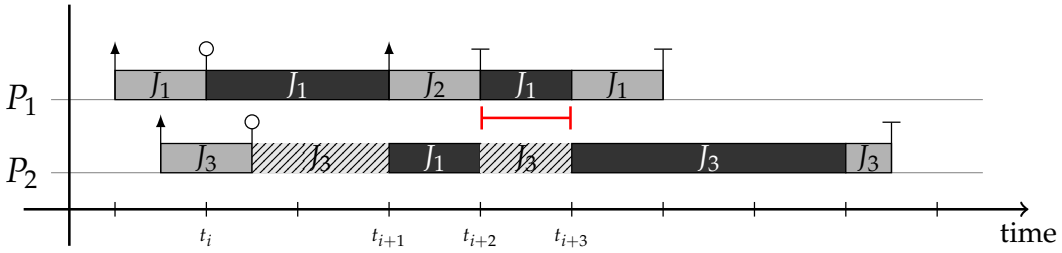
Le soluzioni elencate portano al medesimo risultato, cioè nessun job in P_1 richiede la risorsa, ma con complessità e costi di implementazione ed esecuzione differenti: (a) richiede l'attivazione di un task, rendendo la soluzione la peggiore tra le tre; (b) comporta una migrazione, che, come si vedrà nel capitolo 3, è un'operazione molto onerosa; infine, (c) non aggiunge alcun costo aggiuntivo in quanto lascia un processore inutilizzato diminuendo il costo della primitiva di schedule.

Considerando le soluzioni (b) e (c) da un punto di vista più ampio, la terza soluzione, come si vede in figura 2.6, ritarda la migrazione che avviene al momento del rilascio della risorsa (t_{i+3}). Di conseguenza, (b) e (c) comportano lo stesso numero di migrazioni nell'esempio precedente.

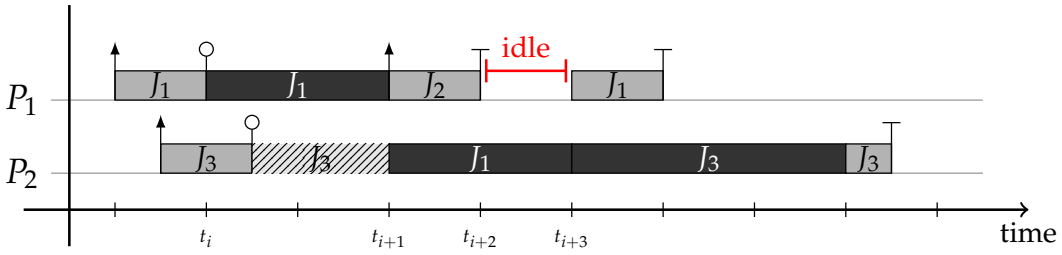
Generalizzando il confronto tra (b) e (c), quest'ultima comporta un numero di migrazioni minore o uguale rispetto alla prima.
Si supponga di utilizzare l'approccio che prevede di migrare nella partizione di partenza: una volta ritornato al processore (sono già avvenute due migrazioni), esso viene prerilasciato dal job J_x e migra nuovamente in una delle partizioni disponibili, per esempio P_3; di conseguenza, sono avvenute quattro migrazioni. Con un approccio basato sulla soluzione (c), vi sono diversi casi possibili da analizzare. Si ricorda che la scelta della partizione in cui migrare avviene in



(a) Attivazione di un agente.



(b) Migrazione nel processore di origine.



(c) Utilizzo del ceiling.

Figura 2.6: Possibili soluzioni alla gestione del caso particolare.

base all'ordine della coda, cioè FIFO; pertanto, il job vaglia sempre come prima opzione il processore in cui è allocato il rispettivo task.

- il job porta a termine la sezione critica nel processore in cui ha migrato in precedenza; (2)
- il job viene prerilasciato e ritorna alla partizione di partenza, in essa porta a termine la sezione critica; (2)
- il job viene prerilasciato e ritorna alla partizione di partenza, in essa viene prerilasciato da J_x e migra in P_3; (4)
- il job viene prerilasciato e, dato che nella partizione di partenza sta eseguendo J_x, migra in P_3. (3)

Nei primi due casi elencati avvengono due migrazioni, nella terza tre e nella quarta quattro. Di conseguenza, si può affermare che l'approccio (c) comporta un numero minore o uguale di migrazioni rispetto a (b). Come detto in precedenza, la migrazione di un job da un processore a un altro è il costo principale che il protocollo comporta, quindi è necessario effettuarne il meno possibile.

L'implementazione del protocollo (righe 25 - 45) utilizza la struttura `mrsp_state` per tenere traccia del livello del ceiling attualmente in vigore in un processore per poter determinare se il processore deve restare inutilizzato, simulando quindi la presenza del job che detiene la risorsa, o eseguire il job a priorità più alta tra quelli nella coda ready.

```
1  task_struct* pfp_schedule(struct task_struct * prev)
2  {
3      int lock_holder, placeholder;
4
5       [...]
6
7      // Determino se il job in esecuzione e' in possesso della risorsa
8      if (prev == local_domain->mrsp->owner) {
9          lock_holder = 1;
10     }
11
12     spin_lock(local_domain->lock);
13
14          [...]
15
16         // Se e' stato prerilascio il possessore della risorsa attivo il flag per la migrazione
17         if (prev && preempt && lock_holder) {
18                 prev.cpu = MIGRATION;
19         } else {
```

```
20          // Altrimenti, se necessario, lo riaccodo nella coda ready
21          if (prev && !blocks && !migrate)
22              requeue(prev, local_domain);
23      }
24
25      struct task_struct *task_head = fp_prio_peek(local_domain->ready_queue);
26
27      // Confronto il job in testa alla coda ready con il ceiling attuale
28      if (task_head->priority > mrsp_state.cpu_ceiling) {
29          placeholder = 0;
30      } else {
31          placeholder = 1;
32      }
33
34       [...]
35
36      // Eseguo il job in testa alla coda ready ready o lascio il processore inattivo
37      if (placeholder == 0) {
38          next = fp_prio_take(local_domain->ready_queue);
39      } else {
40          next = NULL;
41      }
42
43    [...]
44
45    spin_unlock(local_domain->lock);
46
47    return next;
48 }
```

2.3.6 Context switch

L'operazione di *context switch* è richiamata dal plugin quando un job (`prev`) smette di eseguire a favore di un altro `local_domain->scheduled`. Al contrario dell'operazione di schedule, essa non necessita dello spinlock del dominio locale per eseguire, quindi si presta per effettuare le migrazioni.

I meccanismi di migrazione implementati sono tre e servono a gestire situazioni differenti:

MIGRATION. La sezione precedente (2.3.5) discute il funzionamento della primitiva di schedule, la quale ha il compito di notificare al protocollo il pre-rilascio del possessore della risorsa tramite l'utilizzo del flag `MIGRATION`. La primitiva `pfp_finish_switch` gestisce la migrazione vera e propria (righe 9 - 24): dopo aver ottenuto lo spinlock della risorsa, viene cercata una cpu disponi-

bile per l'esecuzione del job. La funzione `find_cpu_available`, già utilizzata in precedenza, scorre la lista delle richieste accodate in ordine FIFO e per ogni nodo controlla lo stato della CPU, la quale è disponibile se o non vi è nessun job in esecuzione, quindi è la CPU di origine del job che possiede la risorsa, o il job che sta eseguendo ha priorità pari al ceiling locale (+1). In caso di successo, il job acquisisce il ceiling della partizione in cui sta per migrare (riga 15).

Meccanismo di base. L'implementazione fornita di P-FP dispone un meccanismo di base per la migrazione. È attivato quando la CPU di un task non è uguale a quella in cui la primitiva è eseguita e, quindi, in cui si trova il job stesso (riga 26). La migrazione avviene come negli altri casi visti finora e si riassume in quattro passaggi: acquisizione spinlock del dominio della partizione, accodamento del task nella `ready_queue`, (se necessario) prerilascio del job attualmente in esecuzione e rilascio dello spinlock. Questo meccanismo viene sfruttato da MrsP all'atto del rilascio della risorsa.

Job resume. Il terzo caso è attivato in presenza di tre circostanze:

- il job in possesso della risorsa è accodato in una qualche partizione (48);
- la CPU in cui esegue la primitiva è tra quelli accodati nella FIFO della risorsa (50);
- la CPU è disponibile per la migrazione del job (55).

Il controllo al terzo punto è simile a quanto descritto per la funzione `find_cpu_available`. La migrazione avviene come nei casi precendenti, quindi acquisendo gli spinlock per togliere il job da una coda ready per poi inserirlo in quella di destinazione. Il controllo a riga 82 è necessario in un sistema parallelo: il possessore della risorsa potrebbe essere tornato in esecuzione nel processore in cui era accodato, oppure un altro processore ha attivato altri meccanismi di MrsP che ne hanno causato la migrazione.

```
1  void pfp_finish_switch(struct task_struct *prev)
2  {
3      mrsp = local_domain->mrsp;
4
5      // Il flag per la migrazione e' attivo?
6      if (prev->cpu == MIGRATION) {
7
8          // Ricerca di una cpu disponibile in cui migrare
9          spin_lock(mrsp->lock);
10             cpu = find_cpu_available(mrsp);
```

```
11      spin_unlock(mrsp->lock);
12
13      // Cambio di cpu e ceiling
14      if (cpu != NULL) {
15          prev->cpu = cpu;
16          prev->priority = mrsp->ceilings[cpu] + 2;
17      }
18
19      // Accodo il job nella coda ready e prerilascio il job che sta effettuando attesa attiva
20      spin_lock(remote_domain->lock);
21          requeue(prev, remote_domain);
22          if (fp_preemption_needed(remote_domain->ready_queue, remote_domain->scheduled))
23              preempt(remote_domain);
24      spin_unlock(remote_domain->lock);
25
26  } else if (prev->cpu != local_cpu) {
27
28      // Meccanismo di base di LITMUS per la migrazione, il parametro cpu e' stato modificato in
            precedenza
29
30      spin_lock(remote_domain->lock);
31          requeue(prev, remote_domain);
32          if (fp_preemption_needed(remote_domain->ready_queue, remote_domain->scheduled))
33              preempt(remote_domain);
34      spin_unlock(remote_domain->lock);
35
36  } else {
37
38      //Caso in cui il job lock holder ha dovuto migrare in un altra cpu, nella quale e' stato a sua
            volta prerilasciato e si trova accodato. Il job che ne aveva
39      //causato la migrazione ha completato il proprio ciclo, la cpu e' tornata disponibile. Quindi o
            non c'e' nessun job schedulato (e' la cpu home del lock
40      //holder?) o e' stato ri-selezionato per eseguire un job che sta effettuando busy wait.
41
42      struct task_struct* owner = NULL;
43      int from_cpu, target_cpu;
44
45      spin_lock(mrsp->lock);
46
47          // Se il lock holder e' accodato...
48          if (is_queued(mrsp->owner)) {
49              // ... e la cpu corrente e' tra quelle insite nella FIFO della risorsa globale
50              if (cpu_queued()) {
51
52                  // Per essere disponibile alla migrazione:
53                  // - o non vi e' nessun job in esecuzione
54                  // - o il job in esecuzione e' quello che sta effettuando attesa attiva
55                  if (local_domain->scheduled == NULL || (local_domain->scheduled->priority ==
                        (mrsp_state.cpu_ceiling + 1))) {
```

```
56                      owner = sem->owner;
57                      from_cpu = owner->cpu;
58                      target_cpu = local_domain->cpu;
59                  }
60              }
61          }
62
63      spin_unlock(mrsp->lock);
64
65      // E' necessario effettuare una migrazione?
66      if (owner != NULL) {
67          preempt_disable();
68
69              bool fail = false;
70
71              spin_lock(from_cpu->lock);
72
73                  if (is_queued(owner)) {
74                      owner->cpu = target_cpu;
75                      fp_dequeue(from_cpu, owner);
76                  } else {
77                      fail = true;
78                  }
79
80              spin_unlock(from_cpu->lock);
81
82              if (! fail ) {
83                  spin_lock(target_cpu->lock);
84                      owner->priority = mrsp->ceilings[cpu] + 2;
85                      requeue(owner, target_cpu);
86                      preempt(target_cpu);
87                  spin_unlock(target_cpu->lock);
88              }
89
90          preempt_enable();
91      }
92  }
93 }
```

2.3.7 Esempio di esecuzione

Questa sezione propone un esempio di esecuzione dello scheduler P-FP e di gestione dell'accesso alle risorse con il protocollo MrsP. Il taskset è creato appositamente per andare a innescare i meccanismi discussi in questo capitolo:

P_1: τ_1 ($prio = 2$), τ_2 (4), τ_3 (7); τ_2 necessita della risorsa, quindi il ceiling locale è pari a 4.

P_2: τ_4 (*prio* = 1), τ_5 (3), τ_6 (8); τ_4 e τ_5 necessitano della risorsa, il ceiling locale è pari a 3.

P_3: τ_7 (*prio* = 2), τ_8 (6); τ_7 necessita della risorsa, il ceiling locale è pari a 2.

P_4: τ_9 (*prio* = 3), τ_10 (7); τ_9 necessita della risorsa, il ceiling locale è pari a 3.

I job che accedono la risorsa eseguono senza di essa sia prima che dopo la sezione critica, quindi il rilascio della risorsa non coincide con il loro completamento.

L'esecuzione illustrata in figura 2.7 rappresenta un esempio di funzionamento del protocollo MrsP nella gestione della contesa per la risorsa, in particolare, evidenzia come le primitive discusse in questo capitolo vadano a operare sulla coda, quindi il fulcro dell'implementazione. Di seguito sono discussi uno per uno gli interleaving interessanti dell'esecuzione e i meccanismi che li gestiscono.

t_1) I job J_9, J_2 e J_4 richiedono la risorsa nel medesimo istante, essi, come visto nella sezione 2.3.3, acquisiscono il ceiling del corrispettivo processore (J_9 +2 mentre J_2 e J_4 +1) e innalzano il ceiling locale. Inoltre accodano le rispettive richieste nella coda FIFO in ordine J_9, J_2, J_4, pertanto J_9 acquisisce la risorsa e gli altri due effettuano attesa attiva.

t_2) J_9 rilascia la risorsa, quindi ripristina la propria priorità il ceiling locale e toglie la propria richiesta dalla coda, inoltre, dato che non vi sono altri job nel medesimo processore, prosegue a eseguire fino al completamento (sezione 2.3.4. Al primo ciclo utile di attesa attiva, J_2 rileva che la propria richiesta ha raggiunto la testa della coda, di conseguenza, acquisisce la risorsa e innalza la propria priorità al valore di ceiling +2. Nel processore P_3, J_7 richiede la risorsa.

t_3) In P_1 viene rilasciato J_3 e, dato che ha priorità superiore rispetto a J_2 (rispettivamente 7 e 6), inizia a eseguire, causando la migrazione del job nel primo processore in coda disponibile (sezione 2.3.6, cioè P_2, in cui acquisisce il valore di ceiling +2 in modo da prerilasciare il job che sta effettuando attesa attiva (J_4). Inoltre, al completamento da parte di J_3, il processore rimane inutilizzato nonostante vi sia J_1 nella coda ready, in quanto quest'ultimo ha priorità inferiore rispetto al ceiling (sezione 2.3.5.

t_4) J_2 termina l'esecuzione della sezione critica mentre si trova in P_2, quindi, dopo aver ripristinato la propria priorità il ceiling in P_1 e tolto la richiesta dalla coda, migra nel processore di origine. In P_1 è ancora il job a priorità superiore, quindi completa l'esecuzione per poi lasciare il processore a J_1. All'atto del rilascio della risorsa, il protocollo rileva che il prossimo possessore della risorsa, J_4, non è in esecuzione, ma è accodato nella medesima partizione. Di

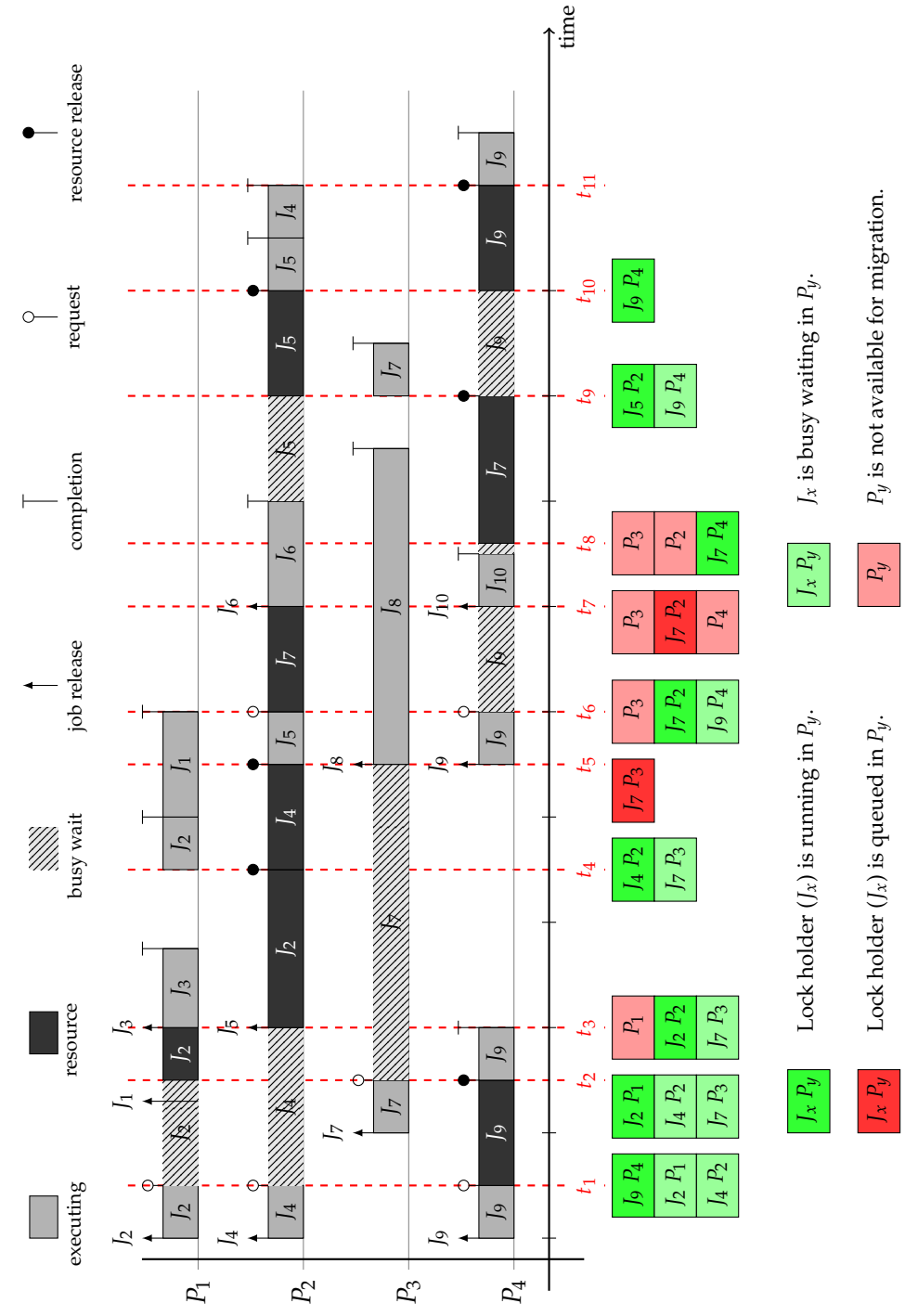


Figura 2.7: Attivazione di un agente.

conseguenza, al termine della primitiva, J_4, tramite l'attesa attiva, acquisisce autonomamente la risorsa.

t_5) J_4 porta a compimento la sezione critica. Il protocollo, tramite la primitiva di rilascio della risorsa, rileva che J_7, il prossimo possessore, è accodato, ma non vi sono processori disponibili per la migrazione. Inoltre, J_4, una volta ripristinata la propria priorità e il ceiling, non è più il job a priorità superiore nella partizione, quindi forza la chiamata alla primitiva di schedule per cedere l'esecuzione a J_5.

t_6) I job J_5 e J_9 richiedono la risorsa, ma J_7 (il job che la detiene) è accodato, quindi J_5, ancor prima di iniziare a effettuare attesa attiva, cede l'esecuzione in P_2 a J_7.

t_7) J_7 è nuovamente prerilasciato. Dato che anche J_9 non è in esecuzione, non vi sono processori disponibili per la migrazione, pertanto viene inserito nella coda ready del processore in cui iera migrato in precedenza.

t_8) In P_4, il job a priorità superiore rispetto al ceiling (J_{10}) completa la sua esecuzione. La primitiva di schedule designa per J_9 per l'esecuzione, il quale riprende a effettuare attesa attiva. La primitiva di context switch, successiva alla schedule, rileva che il lock holder, J_7, è accodato e gli cede l'esecuzione nel processore tramite una migrazione. Successivamente, in P_2, J_5 torna a eseguire attesa attiva dato che il possessore della risorsa non è stato prerilasciato.

t_9) J_7 termina la sezione critica, rilascia la risorsa e migra in P_3, nel quale completa la propria esecuzione in quanto unico job attivo. J_5 acquisisce la risorsa e J_9 ricomincia a effettuare attesa attiva.

t_{10} e t_{11}) J_9 acquisisce e rilascia la risora, la quale rimane inutilizzata in quanto non vi sono job in coda che la necessitano.

Capitolo 3

Esperimenti e valutazioni

Gli esperimenti eseguiti hanno lo scopo di valutare l'implementazione proposta di *MrsP* da diversi punti di vista.

In un primo insieme di esperimenti il protocollo è messo a confronto con altri due protocolli che, come MrsP, sono sviluppati su sistemi partizionati con condivisione di risorse globali: il primo è basato su un approccio *simple ceiling*, mentre il secondo utilizza inibizione di prerilascio. Questo esperimento conside le prestazioni dei protocolli nell'esecuzione di taskset creati su misura per confrontare i *response time* dei vari task in specifiche circostanze. Infine i risultati delle elaborazioni sono confrontati con i dati ottenuti dalle simulazioni in Burns et al. [8].

In seguito, è discussa una valutazione di costi e prestazioni dell'implementazione: una serie di campionamenti permette di verificare il costo aggiunto da MrsP alle primitive dello scheduler, in particolare il tempo computazionale aggiunto per integrare il protocollo in P-FP.

Infine, risulta interessante il funzionamento dello scheduler in assenza di risorse, questo in quanto la presenza di risorse globali condivise in un sistema partizionato è un caso particolare di esecuzione, quindi un buon funzionamento in sua assenza risulterebbe positivo ai fini di una valutazione completa. A tal fine, il sistema è confrontato con P-FP, cioè il medesimo scheduler privo di integrazione con MrsP.

3.1 Ambiente di esecuzione

Gli esperimenti sono effettuati su una macchina fisica dotata di piattaforma i7-2670QM, architettura lanciata da Intel nell'ottobre del 2011.

Sandy Bridge è l'architettura alla base del sistema. Esso consiste in un quad-core con frequenza di clock pari a 2.2 GHz, 3.1 GHz in *Turbo mode*. Ogni core possiede due livelli di cache, L1 e L2 di dimensione rispettivamente pari a 64 KB e 256 KB, ed un terzo livello L3 condiviso tra i 4 core con dimensione di 6 MB. La memoria cache utilizza un metodo di gestione creato da Intel e chiamato Smart cache: permette di diminuire il rapporto globale di *cache miss* aumentando così l'efficienza del suo utilizzo. Questa tecnologia prevede che tutti i livelli di cache siano condivisi tra le CPU e gli accessi assegnati in base alle richieste piuttosto che essere dedicati; inoltre, applica la logica *data-fetch* alle operazioni in modo da mettere a disposizione dei dati ancor prima che siano richiesti. La tecnologia **Simultaneous Multi-Threading** permette di raddoppiare il numero di core, trasformando i 4 fisici in 8 logici, tramite l'esecuzione di due thread sul medesimo core. Questa opzione è disabilitata in fase di test; allo stesso modo anche le funzioni di gestione della potenza sono disattivate. Il **bus** interno alla CPU ha una velocità pari a 100 MHz, mentre il bus seriale che permette le comunicazioni tra processori e con il chipset raggiunge i 5 GT/s. I bus per le connessioni tra processori e chipset utilizzano la tecnologia *Quick-Path Interconnect* (QPI), la cui caratteristica principale consiste nel permettere comunicazioni point-to-point tra le varie componenti; questo è indubbiamente un vantaggio rispetto all'utilizzo del bus come canale unico per tutte le comunicazioni, permettendo così più trasferimenti simultanei.
La piattaforma utilizzata supporta la **Vanderpool Technology**, una particolare tecnologia di virtualizzazione per piattaforme Intel che rende possibile l'esecuzione simultanea di più sistemi operativi ospiti contemporaneamente.

Gli esperimenti sono eseguiti con il supporto di macchina virtuale; l'infrastruttura di virtualizzazione è basata su *Kernel-based Virtual Machine* (**KVM**) che è specifica per i sistemi Linux, mentre il software di emulazione QEMU permette di eseguire il sistema operativo, nel nostro caso l'estensione di Linux con LITMUS^{RT}, come ospite della macchina fisica. L'immagine virtuale utilizzata è in formato compatibile con QEMU ed esegue la bzImage generata tramite la compilazione del kernel.

Il comando di lancio della VM è il seguente:

```
qemu-system-x86_64 -enable-kvm -smp 4 -m 512
-boot c -nographic -net nic -net user,hostfwd=tcp::10022-:22
-kernel bzImage -append console=ttyS0,115200 root=/dev/hda1
-hda ubuntu.backing.qcow2.img
```

Tramite i parametri specificati il sistema di virtualizzazione riconosce che viene lanciato un kernel a 64 bit utilizzando KVM; quest'ultimo supporta la tecnologia di virtualizzazione specifica di Intel denominata Vanderpool citata precedentemente: permette di dividere un sistema in macchine virtuali distinte nonostante condividano le stesse risorse di sistema; ne risultano quindi due macchine logiche, che operano in maniera totalmente indipendente grazie all'appoggio di specifiche funzionalità hardware che consentono di ottimizzare tale condivisione. In particolare, l'architettura hardware abbinata a questa configurazione permette un accesso diretto ai core fisici senza alcun livello di virtualizzazione intermedio.
Alla macchina virtuale vengono assegnati il kernel Linux `-hda`, 4 core fisici e 512 MB di memoria.

In fase di sviluppo sono stati utilizzati alcuni strumenti *user-space* per interagire con LITMUS^{RT}:

- *liblitmus*, Appendice B, è una libreria che permette la creazione ed il controllo di task set;
- `TRACE()` permette di ottenere informazioni dall'esecuzione, è il principale strumento per effettuare debugging.

I campionamenti sono effettuati tramite *Feather-Trace*, il quale consiste in una serie di strumenti atti a calcolare gli overhead delle primitive e rilevare gli eventi di scheduling. Tramite i dati raccolti è stato possibile valutare l'implementazione e confrontare i vari protocolli.
Maggiori informazioni sui metodi di tracciamento e campionamento sono presentati in Appendice C.

3.1.1 Generazione ed esecuzione degli esperimenti

Per la creazione dei taskset sono stati usati tre differenti approcci:

- generati manualmente per ottenere un determinato comportamento;
- tramite applicazione Java per taskset che richiedano l'utilizzo di risorse;
- *experiment-scripts*, una suite di script in Python per la creazione di taskset.

La libreria *experiment-scripts* definisce un formato di file con il quale è possibile avviare, con un unico comando, un intero taskset:

- `sched.py` consiste in una lista di task con relativi parametri (WCET, periodo, risorse, etc.);
- `params.py` contiene informazioni che specificano il plugin utilizzato ed informazioni riguardanti il taskset.

Una volta deciso lo scheduler da utilizzare, il file che specifica il taskset è generato manualmente o tramite script . Maggiori informazioni riguardanti *experiment-scripts* sono riportate in Appendice D.

3.2 Confronto tra protocolli

Nei protocolli *lock-based* l'accesso è gestito, in caso di risorsa occupata, tramite sospensione oppure attesa attiva. Nel primo caso, se la risorsa è occupata, il task richiedente si sospende e viene inserito in una coda in attesa del suo rilascio. Al contrario, in presenza di un protocollo *spin-based*, il richiedente effettua attesa attiva fino al momento di accesso alla risorsa. Tale argomento viene discusso approfonditamente in Brandenburg et al. [5]. MrsP si basa su approccio spin-based, in quanto tale scelta permette di limitare il tempo di blocco subito, ma risulta cruciale come ed in quali circostanze effettuare attesa attiva.

Questo esperimento si sofferma su questo aspetto, cioè in che modo effettuare attesa attiva e a quali condizioni: a seconda del valore di priorità scelto per proseguire l'attesa si ottengono comportamenti differenti che influenzano in modo diverso il sistema.

3.2.1 Esperimento #1

Il seguente esperimento mette a confronto tre protocolli differenti costruiti su sistema partizionato con dispatching basato su priorità e accesso alla risorsa globale gestito tramite accodamento FIFO. In tutti e tre i casi i protocolli sono

basati su SRP: un job inizia a eseguire solamente quando le risorse di cui necessita sono libere, innalza la propria priorità al momento della richiesta ed effettua attesa attiva fino a quando ne ottiene l'accesso esclusivo.

Il protocollo basato su *simple ceiling* prevede che il job innalzi la propria priorità al valore della priorità più alta tra tutti i task allocati nella stessa CPU che la richiedono, esattamente come MrsP. Il suo comportamento è analogo a quello di MrsP, salvo il fatto che non vi è nessun meccanismo di migrazione.

Il terzo protocollo prevede che al momento della richiesta venga inibito il prerilascio localmente alla CPU del richiedente. Questo effetto in fase di implementazione è stato ottenuto innalzando la priorità al valore massimo, non permettendo a nessun job di causare prerilascio.

L'esperimento prevede di mettere a confronto i response time dei task che compongono il sistema, analizzando quali vengano maggiormente penalizzati dall'attesa attiva nei tre protocolli al variare di parametri come la lunghezza della sezione critica o del WCET di determinati task.

3.2.2 Configurazione

Il taskset prevede 3 task (τ_1, τ_2, τ_3) allocati su due CPU: sulla prima CPU viene allocato un task a priorità maggiore e uno a priorità inferiore, quest'ultimo condivide la risorsa globale con un task allocato nella seconda CPU.

L'esecuzione è impostata in modo tale che il primo job a essere rilasciato ed eseguire sia quello a priorità inferiore allocato nella prima CPU (L_1), poi il job della seconda CPU (L_3) e infine quello a priorità più alta (H_2. L'esecuzione voluta è rappresentata nella figura 3.1:

- punto 1, il job a priorità più bassa è rilasciato e ottiene la risorsa in quanto libera;
- punto 2, il secondo job a bassa priorità si accoda ed effettua attesa attiva nella seconda CPU;
- punto 3, Il job a priorità più alta tenta di eseguire causando prerilascio del primo job; quest'ultimo passaggio viene gestito in modo differente dai tre protocolli.

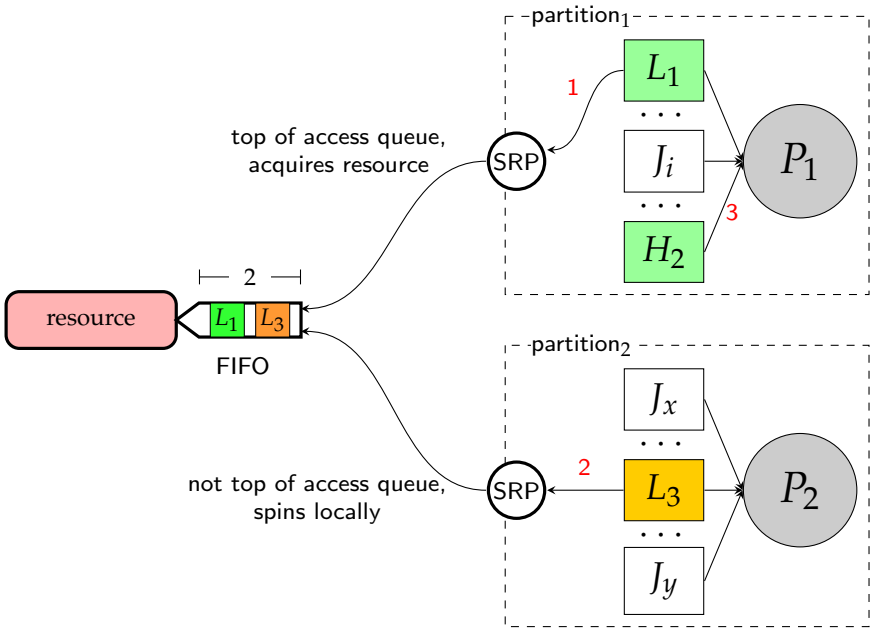


Figura 3.1: Configurazione del test tra protocolli.

3.2.3 Obiettivo

Variando il tempo di esecuzione del job a priorità più alta e la lunghezza della sezione critica della risorsa globale il comportamento atteso è che in ogni protocollo sia differente il job che soffre maggiormente tale cambiamento.

Nei capitoli precedenti è stato chiarito come la condivisione della risorsa tra più CPU in un sistema partizionato vada ad aumentare il costo pagato da alcuni job. In MrsP, tale condivisione influenza solamente i job che la vogliono ottenere e coloro che subiscono blocco da essi, cioè non la richiedono, ma un job a priorità più bassa la contende ad uno a priorità superiore alla propria tra quelli della medesima CPU.

Privando MrsP dei meccanismi per gestire i prerilasci si ottiene un comportamento simile a quello del protocollo basato su *simple ceiling*: i job a priorità maggiore non verranno influenzati, ma al tempo stesso aumenta il tempo di blocco subito dai job a priorità inferiore al ceiling e i tempi di attesa dei job accodati sulla risorsa allocati in altre CPU. Tale aumento di blocco e attesa è determinato dal fatto che il proprietario della risorsa non può proseguire l'esecuzione della sezione critica in quanto prerilasciato. Il risultato è un sistema in cui l'interferenza subita dal lock holder si ripercuote anche sugli altri processori.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	1
L_3	P_2	20	1	1

Tabella 3.1: Confronto tra protocolli: primo task set.

Il comportamento atteso dal protocollo che inibisce il prerilascio è che il job a soffrire maggiormente della condivisione sia quello localmente ready a priorità più alta: esso subisce un ritardo pari all'esecuzione della sezione critica nonostante non necessiti della risorsa globale. Il risultato è quindi che soffrano blocco anche i job che non rispecchiano la condizioni fornite in precedenza, la loro esecuzione di conseguenza viene ritardata fino a che il job non ripristina la proprio priorità al momento del rilascio della risorsa.

Un altro aspetto che si vuole studiare è come le migrazioni vadano a influenzare le prestazioni di MrsP: un ultimo taskset è configurato in modo tale che le circostanze che forzano il job dalla prima CPU alla seconda a migrare si ripresentino anche in quest'ultima, obbligando a una seconda migrazione nella terza CPU in cui eseguire la sezione critica.

3.2.4 Risultati esperimento #1

La tabella 3.1 rappresenta il primo task set. In esso i job hanno i medesimi tempi di esecuzione (pari a un millisecondo). La durata della sezione critica è compresa nell'esecuzione del WCET; nel caso abbiano lo stesso valore si intende che il job effettui la richiesta di accesso come prima azione e completi al momento del rilascio la propria esecuzione. Ne consegue che l'evento di rilascio della risorsa e quello di completamento del task non coincidono. Questo particolare è importante in quanto, in alcuni casi, nelle tempistiche riportate si considera il momento di rilascio della risorsa.

La tabella 3.2 riporta i tempi di completamento di ogni job per i tre protocolli. I valori sottolineati indicano l'istante di rilascio della risorsa piuttosto che il completamento, questo in quanto il resto esecuzione subisce interferenza da parte del job a priorità superiore ed esula dai compiti dei protocolli di accesso a risorsa.

Task	MrsP	Ceiling	Non preemption
L_1	1.206.362	2.194.042	1.111.517
H_2	1.098.587	1.068.602	1.977.039
L_3	2.351.562	3.168.240	1.911.890

Tabella 3.2: Confronto tra protocolli: risultato primo task set, tempi espressi in nano secondi. I valori sottolineati si riferiscono al rilascio della risorsa e non al completamento del job.

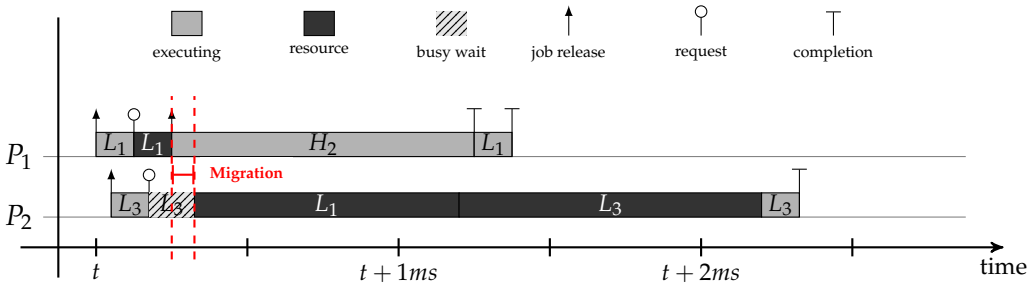


Figura 3.2: *MrsP*.

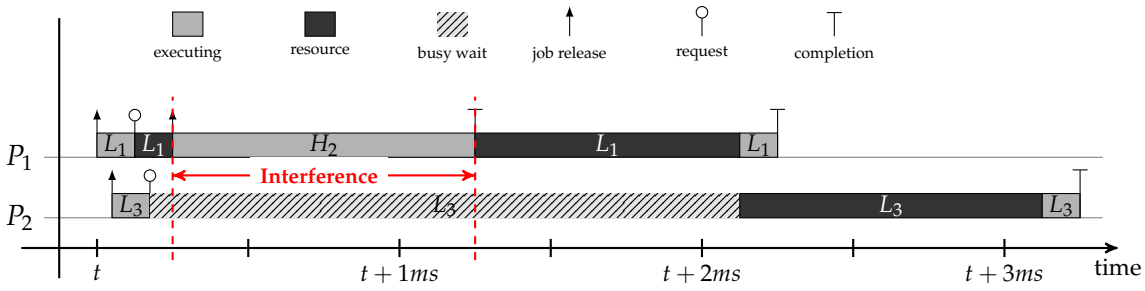


Figura 3.3: *Simple ceiling*.

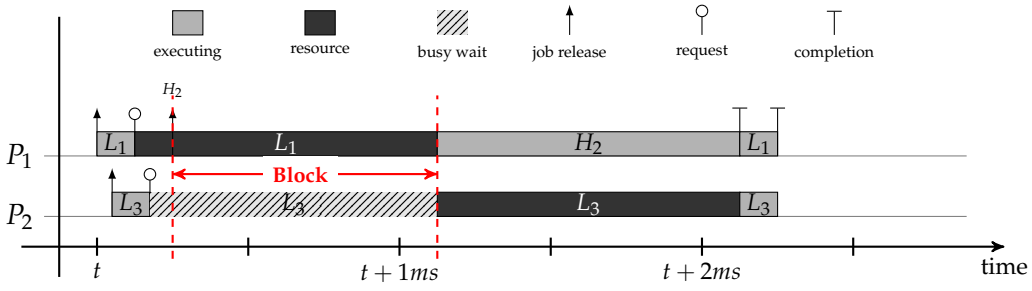


Figura 3.4: *non preemption*.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	3	3
H_2	P_1	10	0	1
L_3	P_2	20	3	3

Tabella 3.3: Confronto tra protocolli: aumento della sezione critica.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	3
L_3	P_2	20	1	1

Tabella 3.4: Confronto tra protocolli: aumento dell'interferenza.

L'esecuzione del taskset è rappresentato graficamente in 3.2, 3.3 e 3.4. Analizzando i dati, si nota come in MrsP la migrazione renda minimo il tempo di attesa subito da L_3 e nullo il blocco subito da H_2. I valori di esecuzione rappresentati subiscono degli overhead dati dal sistema, in particolare il costo della migrazione. Questo comportamento viene messo in risalto nella figura 3.2; le due linee rosse tratteggiate evidenziano l'overhead dato dal cambio di processore. Tale lasco di tempo consiste nel tempo che impiega il job a migrare, per cui L_3 continua a effettuare attesa attiva e lo smaltimento della coda FIFO viene rallentato in quanto L_1 non sta progredendo nell'esecuzione della sezione critica.
Con l'approccio basato su simple ceiling il tempo di attesa di H_2 non dipende più solamente dalla lunghezza della coda della risorsa, ma anche dall'interferenza che il lock holder subisce. La figura 3.3 mostra come l'interferenza penalizzi ogni processore in cui vi sia un job in attesa della risorsa. Modificando il tempo di esecuzione di H_2 ci si aspetta che tale costo aumenti o diminuisca di conseguenza.
Al contrario dei casi precedenti, inibendo il prerilascio il job a soffrire maggiormente la condivisione della risorsa è H_2 in quanto non riesce ad eseguire nonostante non la richieda. L'inizio dell'esecuzione del job a priorità maggiore viene quindi ritardata; in figura 3.4 è evidenziato questo comportamento. Pertanto, H_2 subisce un maggiore blocco all'aumentare della sezione critica.

Nei tasksets 3.3 e 3.4 sono state apportate modifiche rispettivamente alla lunghezza della sezione critica e al tempo di esecuzione del job a priorità maggiore.

Nel primo caso il comportamento è quello che ci si aspetta: la figura 3.5 mostra

Task	MrsP	Ceiling	Non preemption
L_1	3.066.828	4.242.092	3.177.307
H_2	1.035.721	1.141.324	3.956.506
L_3	6.099.752	7.209.873	6.024.691

Tabella 3.5: Confronto tra protocolli: aumento della sezione critica, tempi espressi in nano secondi. I valori sottolineati si riferiscono al rilascio della risorsa e non al completamento del job.

Task	MrsP	Ceiling	Non preemption
L_1	1.053.232	4.215.599	1.113.397
H_2	3.018.344	3.071.190	4.006.309
L_3	2.042.122	5.169.139	2.068.905

Tabella 3.6: Confronto tra protocolli: aumento dell'interferenza, tempi espressi in nano secondi. I valori sottolineati si riferiscono al rilascio della risorsa e non al completamento del job.

come un aumento della sezione critica con MrsP dilunghi il tempo di attesa di L_3, che viene utilizzato come nel caso precedente per far proseguire L_1 al momento del prerilascio, mentre H_2 resta inalterato. Al contrario, con simple ceiling sia L_1 che L_3 subiscono l'interferenza da parte di H_2. Infine, inibendo il prerilascio il job di L_3 subisce ulteriormente l'inversione di priorità.

Agendo sul WCET di H_2 (tabella 3.6) si dimostra come MrsP sia più performante al netto dei costi della migrazione, mentre con simple ceiling la maggiore interferenza subita da L_1 rende maggiore il tempo di attesa di L_3. Inibendo il prerilascio il maggior tempo di esecuzione non influisce sui job che vogliono accedere la risorsa e l'inizio dell'esecuzione di H_2 è posticipato allungando così il tempo di completamento.

Con il taskset illustrato nella tabella 3.7 si intende ripercorrere il funzionamento del primo esempio forzando in questo caso il job che detiene la risorsa ad affrontare 2 migrazioni. Inoltre il fatto che vi siano 3 CPU a contendere per l'accesso della risorsa triplica il fattore per cui la sezione critica viene moltiplicata. A seconda del protocollo utilizzato aumenta l'attesa dei job accodati o il blocco subito dai job a priorità più alta.

Nella tabella 3.8 il comportamento è quello atteso e discusso finora, l'aspetto interessante è vedere come l'esecuzione di circa 1 ms di L_1 sia aumentato ulteriormente da 1.05 ms a 1.1 ms a causa della doppia migrazione.

Task	Partition	priority	Critical section	WCET
L_1	P_1	20	1	1
H_2	P_1	10	0	3
L_3	P_2	20	1	1
H_4	P_2	10	0	3
L_5	P_3	20	1	1

Tabella 3.7: Confronto tra protocolli: doppia migrazione.

Task	MrsP	Ceiling	Non preemption
L_1	1.111.410	4.312.490	1.173.600
H_2	3.029.214	3.131.630	4.205.578
L_3	2.062.770	5.301.240	2.211.347
H_4	3.022.036	3.099.090	5.078.436
L_5	3.030.634	6.309.370	3.184.333

Tabella 3.8: Confronto tra protocolli: doppia migrazione, tempi espressi in millisecondi. I valori sottolineati si riferiscono al rilascio della risorsa e non al completamento del job.

3.2.5 Considerazioni

I risultati esaminati mettono in risalto le differenze tra i vari protocolli nella gestione della risorsa globale. Si nota come MrsP sia migliore rispetto alle alternative qui studiate: simple ceiling è inadatto a gestire la condivisione di risorse in ogni caso analizzato, mentre l'inibizione del prerilascio risulterebbe migliore in uno scenario in cui la lunghezza della sezione critica fosse inferiore rispetto agli overhead dati da MrsP, in particolare il costo della migrazione.

Quest'ultimo aspetto è discusso da Burns et al. [8]: se i costi aggiunti dal sistema sono superiori rispetto alla lunghezza della sezione critica allora il protocollo non è più utile. Tale affermazione è confermata a seguito dell'implementazione e dei test, essi mettono in risalto come la presenza di costi aggiuntivi vada a penalizzarne le prestazioni.

Nei grafici 3.5, 3.6 e 3.7 sono riassunti i risultati ottenuti nell'esperimento, mettendo a confronto i tempi raccolti per ogni task nelle diverse configurazioni del sistema.

In figura 3.5 è preso in considerazione L_1, cioè il task a bassa priorità che richiede e accede la risorsa per primo. Il grafico evidenzia come MrsP e l'inibizione del prerilascio abbiano prestazioni molto simili, salvo il costo della migrazione

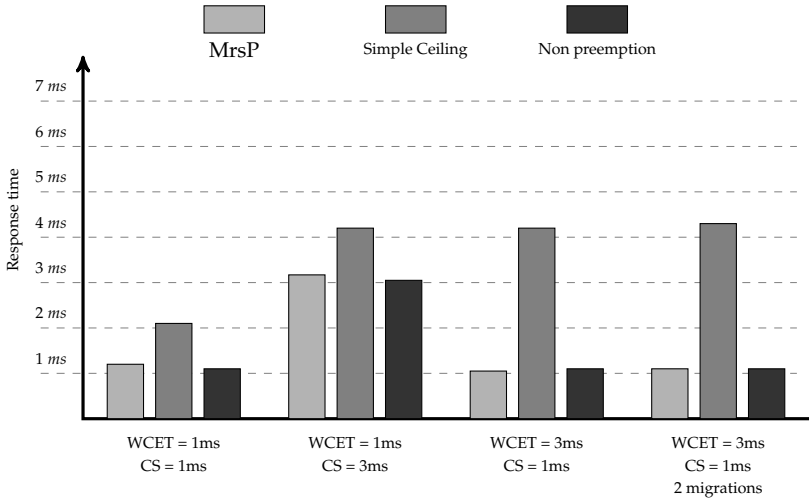


Figura 3.5: Response time di L_1

nel primo caso; al contrario simple ceiling porta a risultati peggiori in ogni configurazione del sistema.

Il grafico 3.6 rappresenta le prestazioni raccolte di H_2: nei sistemi gestiti tramite utilizzo di ceiling i task a priorità superiore non risentono della presenza di risorse, mentre l'inibizione del prerilascio causa notevoli ritardi nei tempi di completamento dei job in quanto subisce blocco dall'esecuzione della sezione critica.

Infine il grafico 3.7 evidenzia come il job L_3 subisca l'interferenza a cui è soggetto L_1: MrsP e non preemption tentano di minimizzare tale interferenza, di conseguenza anche L_3 giova di questi meccanismi anche se con il primo protocollo si notano i costi dovuti dalla migrazione. Nel caso di simple ceiling il job in questione incorre nell'interferenza che avviene nella prima CPU oltre che nei tempi di attesa dovuti dallo smaltimento della FIFO.

L'esperimento illustrato in questa sezione è tratto dai test simulati da Burns e Wellings in [8]. L'implementazione sviluppata e valutata in questo lavoro di tesi rispecchia ciò che gli autori affermano. In particolare si evidenzia come MrsP sia una buona combinazione dei vantaggi degli altri approcci esaminati in questo esperimento. Inoltre, una reale implementazione, al contrario della simulazione, permette di ottenere un migliore riscontro dei costi aggiuntivi dati dalle primitive e, soprattutto, dalle migrazioni.

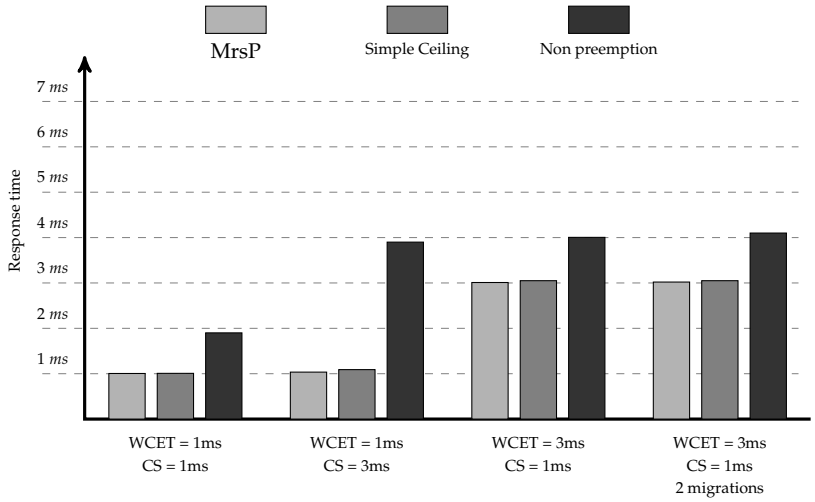


Figura 3.6: Response time di H_2

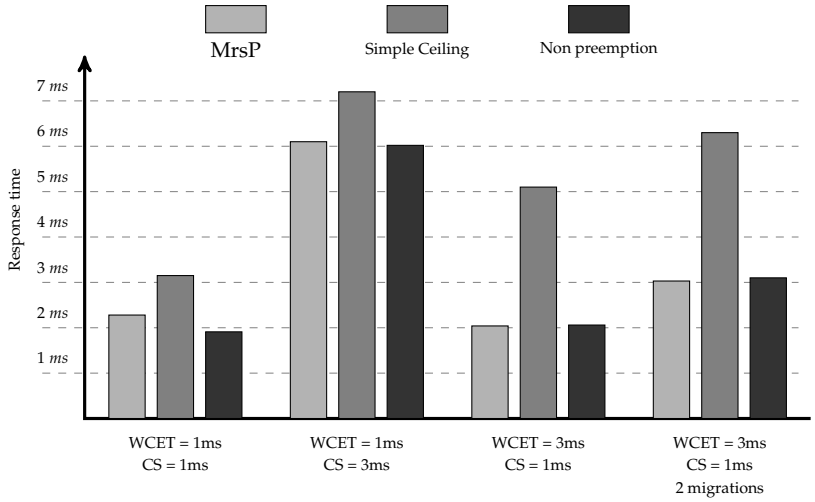


Figura 3.7: Response time di L_3

3.3 Calcolo degli overhead

MrsP combina approcci differenti in parte tratti da protocolli esistenti, i quali, nonostante alcuni aspetti positivi, hanno un funzionamento inapplicabile ad un sistema real-time oppure hanno una complessità che a livello teorico sembra ragionevole ma che in un ambiente reale non è sostenibile. Uno degli obiettivi di questo lavoro di tesi è dimostrare che è possibile implementare il protocollo di Burns et al. [8] a partire da uno scheduler P-FP con un sovraccarico ragionevole del sistema.

3.3.1 Esperimento #2

Gli esperimenti esposti in questa sezione mirano a valutare l'implementazione e le scelte algoritmiche, i relativi overhead vengono studiati per capire quali task ne risentano e in quali circostanze.

Lo sviluppo del protocollo a partire dall'implementazione fornita da LITMUS^{RT} di P-FP ha reso necessario modificare alcune primitive per integrare il protocollo di accesso. Quelle prese in considerazioni sono le seguenti:

- creazione della risorsa;
- richiesta di accesso;
- rilascio della risorsa;
- chiusura della risorsa;
- operazione di schedule;
- *finish-switch*.

3.3.2 Configurazione

Il sistema non permette di avere un release time unico per tutti i job a meno di creare un taskset caratterizzato da periodi armonici, ma in questo caso risulterebbe difficile riuscire a testare alcuni meccanismi che solamente in casi particolari entrano in gioco. In un sistema privo di periodi armonici i task sono soggetti a *release latency*, di conseguenza non è possibile forzare determinate dinamiche con precisione. Per ottenere campionamenti per ogni meccanismo che caratterizza MrsP è necessario creare un taskset sufficientemente grande in modo che si vengano a creare naturalmente le circostanze per poterne usufruire. Il taskset è generato randomicamente con 25 job suddivisi su 4 CPU in modo il più possibile bilanciato. Per ogni CPU sono selezionati alcuni job ai quali è

aggiunta la richiesta alla risorsa condivisa. Ogni taskset, prima di essere eseguito, è analizzato con uno script creato per verificare che sia *feasible* andando ad applicare la *response time analysis* aumentata con il protocollo MrsP.

3.3.3 Obiettivo

Lo scopo di questo esperimento è valutare l'impatto del protocollo in termini di overhead; i campionamenti sono messi in relazione con i tempi di esecuzione ed i costi aggiunti dal sistema.

Non tutte le primitive sono interessanti per lo studio dei costi che il protocollo aggiunge a *runtime*: la creazione e la chiusura non sono presi in considerazione in quanto eseguite in fase di inizializzazione e finalizzazione della risorsa.

L'operazione di scheduling subisce un'unica modifica che permette di bloccare la coda dei job ready quando la richiesta in testa a tale coda ha priorità inferiore rispetto al ceiling locale. Ne consegue che il costo della primitiva risulta uguale o minore rispetto alla versione originale, di conseguenza, è poco interessante per i campionamenti.

Più complesse sono le primitive inerenti a **lock** e **release** della risorsa e l'operazione di **finish-switch**; quello che ci si aspetta è che l'overhead aggiunto in caso di esecuzioni nella norma non sia elevato e che vada ad aumentare a seconda dei meccanismi attivati. Per esecuzione normale si intende quei casi in cui non vi sono interferenze al lock holder da parte di job a priorità superiore al ceiling locale, pertanto i job che contendono per il possesso della risorsa effettuano la richiesta, se occupata attendono fino ad ottenerla, eseguono la sezione critica e la rilasciano.

I costi rilevati durante i campionamenti possono influenzare il sistema a tre differenti livelli (figura 3.8):

- (i) il solo job che esegue la primitiva: questo accade se non si va a modificare il ceiling locale, di conseguenza non si causa blocco ai job a priorità inferiore a tale valore;
- (ii) la sola CPU in cui è allocato il job corrente: questo succede nei casi in cui i costi vengano generati in circostanze di ceiling innalzato ma non si è in possesso della risorsa;

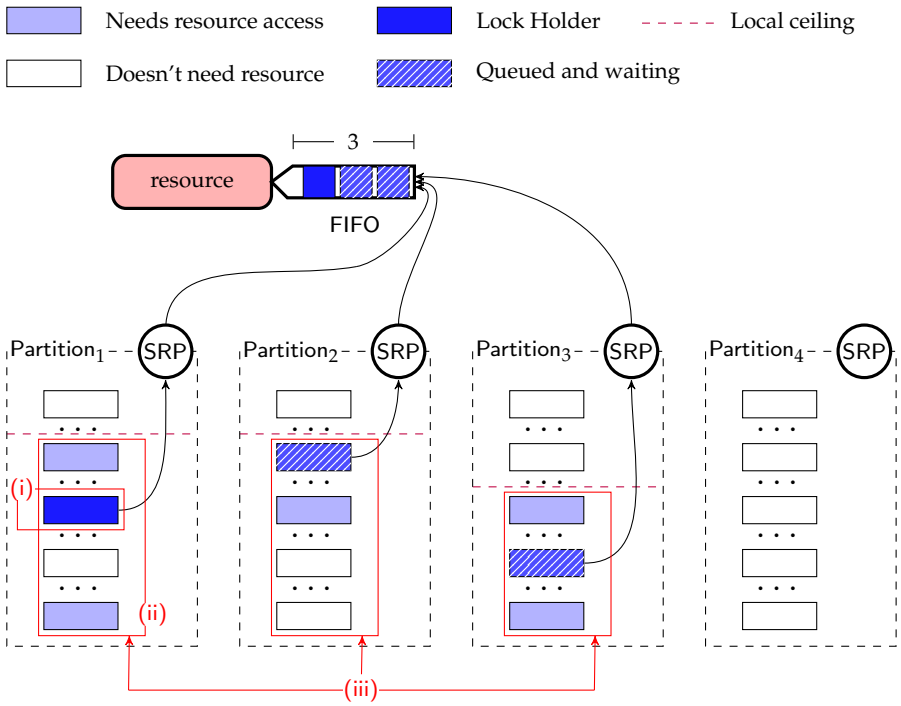


Figura 3.8: Overhead e relativa influenza sul sistema.

- (iii) l'intero sistema, inteso come le CPU in cui stanno eseguendo job che attendono di accedere la risorsa.

3.3.4 Risultati esperimento #2

I dati del sistema sono organizzati secondo la figura 3.9: per ogni CPU è presente una struttura dati che tiene traccia del task attualmente in esecuzione, il numero della CPU, la coda dei task ready in attesa di eseguire ed il ceiling locale; la risorsa invece è composta dal task che la detiene, la coda di task in attesa di ottenerla e la lista dei ceiling calcolati in fase di inizializzazione. L'accesso alle strutture dati da parte dei processi viene reso sequenziale ed in mutua esclusione tramite l'uso dei **spinlock**. Quest'ultimo consiste in un particolare tipo di lock del kernel Linux, lo si acquisisce tramite l'operazione `spin_lock(lock)` e lo si rilascia con `spin_unlock(lock)`. Di conseguenza le primitive che prevedono l'utilizzo delle strutture dati prima di eseguire la sezione critica devono ottenere lo spinlock, tale operazione aggiunge ulteriori overhead oltre a quelli dettati dal protocollo in sé.

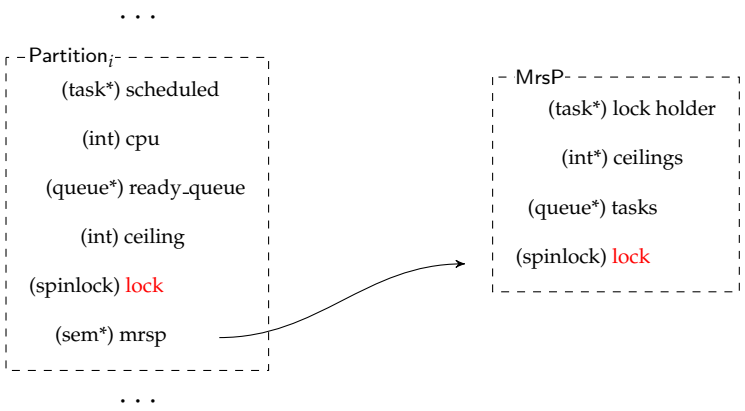


Figura 3.9: Strutture dati.

Di seguito, sono prese in considerazione le primitive, per ognuna si indicando gli overhead campionati; i valori rilevati sono espressi in nanosecondi.

Lock L'operazione di richiesta della risorsa consiste nei tre passaggi elencati di seguito e raffigurati in figura 3.11:

1. innalzamento della priorità del job e del ceiling locale al valore calcolato in fase di inizializzazione, cioè la priorità più alta tra tutti i job allocati in quella CPU che accedono la risorsa, accodamento della richiesta nella FIFO ed infine determina la stato corrente della risorsa ed eventuale proprietario;
2. in base alle informazioni ottenute al passo precedente, se il proprietario non è in esecuzione, cioè è accodato nella coda dei job ready di un altro processore, gli viene concesso di eseguire nella CPU corrente, viene perciò effettuata una migrazione;
3. se il job che ha inoltrato la richiesta non ha ottenuto la risorsa effettua attesa attiva fino ad arrivare in testa alla coda FIFO.

La prima fase causa un overhead pari a circa **800 ns** e costituisce il costo principale riguarda le operazioni sulla coda di richieste. Questo costo è intrinseco alla risorsa e viene pagato dal job stesso e da quelli della stessa CPU. Tale influenza è motivata dal fatto che la prima operazione che viene effettuata è l'innalzamento di priorità e del ceiling, di conseguenza il tempo di blocco oltre alla sezione critica comprende anche questo overhead.

I dati utili a questa operazione sono condivisi con altri processi, di conseguenza l'accesso deve essere gestito tramite il sistema di spinlock in modo da garantire

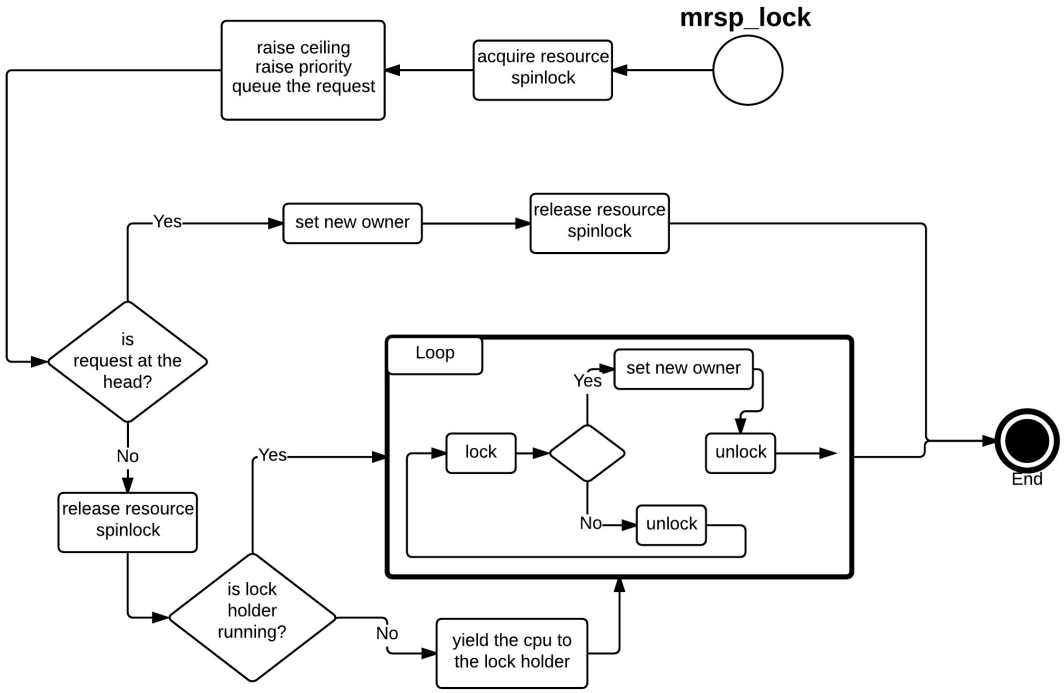


Figura 3.10: Lock: diagramma di flusso.

mutua esclusione ed evitare stati inconsistenti causati da accessi in parallelo da distinte CPU. Per eseguire i passaggi indicati in questa fase è necessario quindi aver acquisito lo spinlock della risorsa, aumentando il tempo necessario per effettuare la richiesta di accesso. Il suo costo va ad influire solamente sul job in questione ed il suo ammontare dipende dal numero di processori che contendono: un maggior numero di richieste in parallelo significa maggior interferenza, quindi un incremento dell'attesa.

La seconda fase consiste nel togliere il job dalla coda ready della CPU in cui si trova per modificarne la priorità ed accodarlo in quella corrente. In questo modo è possibile selezionarlo per l'esecuzione alla successiva operazione di scheduling. Le due singole operazioni sulle code richiedono in media un tempo pari a **500 ns**. Anche in questo caso il sistema è dotato di un sistema di lock che mirano a serializzare le operazioni, quindi è necessario ottenere lo spinlock prima di una CPU e poi, dopo aver rilasciato il precedente, di quella corrente. Le misurazioni del tempo complessivo della seconda fase indicano che le tempistiche medie sono vicine ai **2k ns**, e rispecchiano la contesa per ottenere il lock sulle CPU.

Il costo della migrazione è in media pari a **6k ns**, esso consiste nel tempo che impiega la CPU corrente ad effettuare il context switch tra il job che richiede la risorsa e quello che la detiene. Tale cambio di job in esecuzione viene effettuato tramite un'operazione di scheduling; i strumenti di misurazione delle primitive di sistema confermano che il costo complessivo della migrazione è composta dai singoli costi delle primitive richiamate per togliere il job dall'esecuzione in un processore e trasferirla sul processore di destinazione.

Gli overhead identificati influiscono sull'intero sistema in quanto l'esecuzione della sezione critica da parte del job che detiene la risorsa non riprende immediatamente non appena vi è una nuova CPU disponibile, bensì dopo una quantità di tempo determinata principalmente dalla migrazione.

La terza ed ultima fase della primitiva di richiesta della risorsa consiste nell'eseguire attesa attiva. Essa consiste nell'eseguire ciclicamente un controllo alla testa della coda FIFO, e nel caso sia il turno del job corrente acquisisce la risorsa. Ogni ciclo ha un costo di **500 ns**, anche in questo caso dettato dalle operazioni sulla coda; questo dato non tiene conto dell'overhead per ottenere lo spinlock della risorsa.

Quest'ultimo caso è un costo intrinseco del protocollo e dell'approccio spin-based, di conseguenza non aggiunge alcun costo, salvo un eventuale offset dato dal tempo che il job impiega ad accorgersi che è il suo turno. In tale circostanza si va ad allungare i tempi di attesa degli altri job contendenti, quindi anche il tempo di blocco subito nelle CPU.

Relsease Al momento del rilascio della risorsa vengono effettuate le seguenti operazioni:

1. il job rilascia la risorsa, ripristina la propria priorità ed il ceiling della CPU in cui è allocato e toglie la richiesta dalla testa della coda;
2. se la FIFO non è vuota, viene controllato lo stato del job in testa, se non sta eseguendo si cerca una CPU in cui farlo eseguire;
3. il job rilascia la CPU in cui sta eseguendo se non si trova nella propria di origine.

Le operazioni della prima fase causano un overhead di **500 ns**; come nelle circostanze precedenti ad influire sono le manipolazioni della FIFO. In questo modo si posticipa l'esecuzione del prossimo job in testa alla coda, di conseguenza influenza ogni CPU in attesa di accesso.

Come nel caso dell'acquisizione della risorsa, in alcune circostanze è necessario forzare il prossimo lock holder a migrare in un'altra CPU per proseguire nell'esecuzione e limitare i tempi di attesa della coda FIFO. I campionamenti hanno evidenziato le medesime tempistiche: **2k ns** per le operazioni eseguite tenendo in considerazione i tempi causati dai spinlock sulle CPU e **6k ns** per effettuare la migrazione vera e propria. L'overhead viene pagato dall'intero sistema.

Nell'ultima fase il job che ha rilasciato la risorsa, se necessario, ritorna alla propria CPU di origine. Questo viene effettuato andando ad abbassare la priorità del job al di sotto del ceiling e forzando lo scheduler a selezionare la testa della coda ready.

Tale operazione raccoglie gli overhead delle seguenti operazioni:

1. parte della chiamata di sistema per rilasciare la risorsa;
2. schedule nella CPU attuale;
3. operazione di post-schedule, la quale riaccoda il task nella propria CPU;

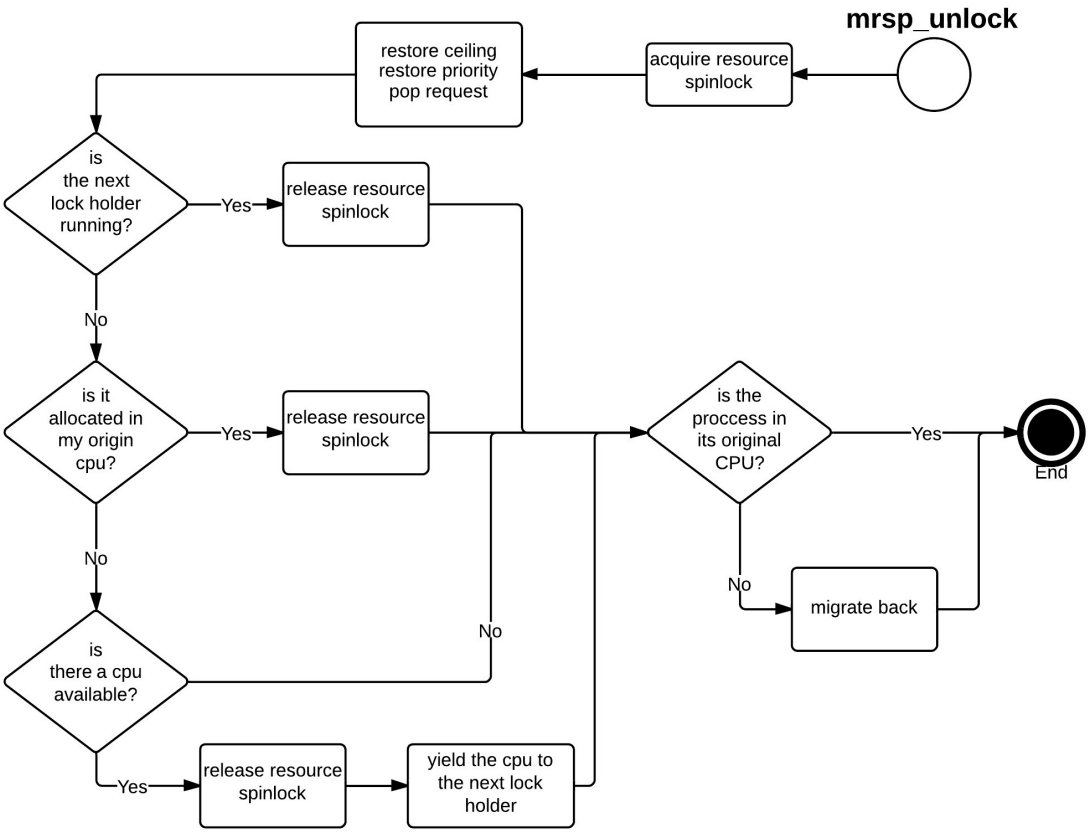


Figura 3.11: Lock: diagramma di flusso.

4. schedule nella CPU di origine.

Questa operazione di conseguenza risulta molto costosa, circa **65k ns**, è pagata solamente dal job che ha effettuato il rilascio. Tale attribuzione consegue dal fatto che la risorsa già nella prima fase è stata rilasciata, quindi potenzialmente il prossimo job può eseguire la propria sezione critica, ed il valore di ceiling della CPU di origine è stato ripristinato, permettendo ai job a priorità inferiore di eseguire nuovamente.

Finish-switch Al contrario delle altre primitive, non vi è una sequenza di fasi, bensì, se necessario, si aziona solamente un meccanismo tra quelli implementati nella primitiva:

- meccanismo base di migrazione di MrsP in caso di prerilascio;
- meccanismo di notifica di CPU di nuovo disponibile
- meccanismo di migrazione di LITMUS^{RT}

Il primo prevede di cercare una CPU disponibile per la migrazione, il suo costo dipendente principalmente dalla lunghezza della coda. In caso di successo, il job modifica la propria CPU di riferimento e la priorità in base al nuovo ceiling, infine viene accodato nella CPU disponibile, forzando su di essa un'operazione di scheduling. Questo insieme di operazioni risulta onerosa in quanto opera sulla coda e necessita dello spinlock della risorsa e della coda ready. Il campionamento ha evidenziato un caso medio di circa **24k ns**.

Inoltre tali operazioni generano una migrazione: al contrario dei casi precedenti il job non è accodato, bensì è in uno stato di prerilascio. Il costo campionato è pari a **37k ns**.

Nel secondo caso viene inizialmente elaborato lo stato della CPU e del lock holder, se quest'ultimo non è in esecuzione e la CPU corrente è tra quelle accodate nella FIFO della risorsa si ricorre alla migrazione per cedere l'esecuzione. I campionamenti indicano che la prima parte ha un costo di **3k ns**. Questo in quanto si necessita di mutua esclusione durante la manipolazione della coda, quindi di acquisire lo spinlock. Il costo della migrazione è in linea con gli altri casi in cui il job preso in considerazione è accodato: circa **6k ns**.

Gli overhead campionati che caratterizzano la primitiva di finish-switch affliggono il job mentre detiene la risorsa, quindi i ritardi provocati penalizzano l'intero sistema.

3.3.5 Considerazioni

Questo esperimento e gli overhead riportati evidenziano come MrsP abbia un costo relativamente basso nella maggior parte dei casi, cioè quando non vi sono circostanze in cui i vari meccanismi di migrazione entrano in gioco.

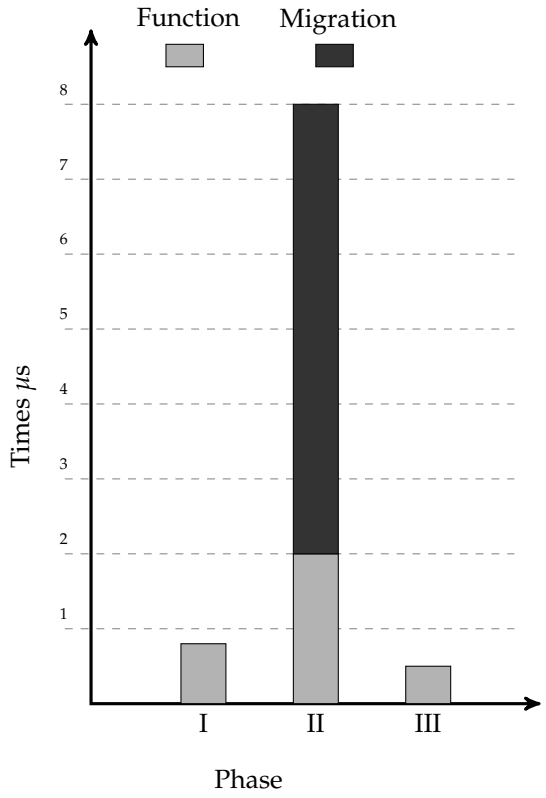
I strumenti di misurazione permettono di avere un campionamento delle operazioni nella loro interezza, cioè da quando viene effettuata la *system call* a quando si è conclusa. Tali misurazioni hanno evidenziato come le primitive di LOCK e UNLOCK della risorsa comportino un overhead di circa **2k ns**, di cui solamente **500 / 800 ns** sono riportabili all'implementazione di MrsP. Questo permette di affermare nuovamente che se non vi sono prerilasci il protocollo ha un impatto relativamente basso sul sistema.

Considerazioni differenti vanno fatte in caso di prerilascio: l'operazione più onerosa è data dalla migrazione al momento del rilascio della risorsa, essa accade solamente una volta ed il suo overhead rispecchia come si vadano ad accumulare una serie di costi di sistema e relative primitive prima che il job riprenda ad eseguire nella propria CPU.

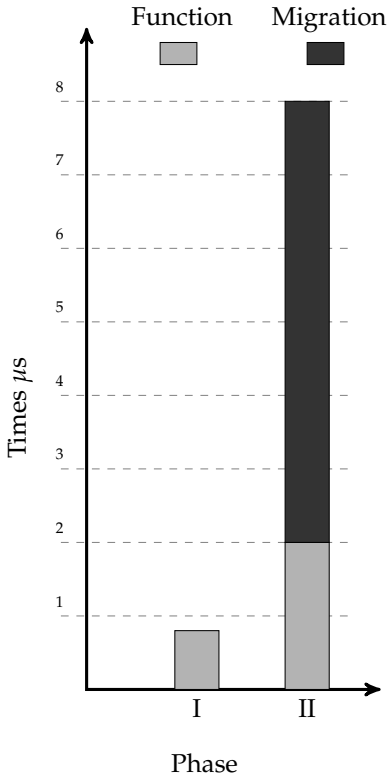
Gli altri meccanismi di migrazione al contrario sono meno onerosi, hanno però lo svantaggio che non si può stimare quante volte verranno innescate nell'arco di una sezione critica dato che dipendono dalle dinamiche del sistema durante l'esecuzione della sezione critica.

Le tempistiche campionate sono rappresentate graficamente nel grafico 3.12, in esso viene tralasciata la migrazione al momento del rilascio in quanto affligge solamente il job che la deve effettuare senza influenzare altri task.

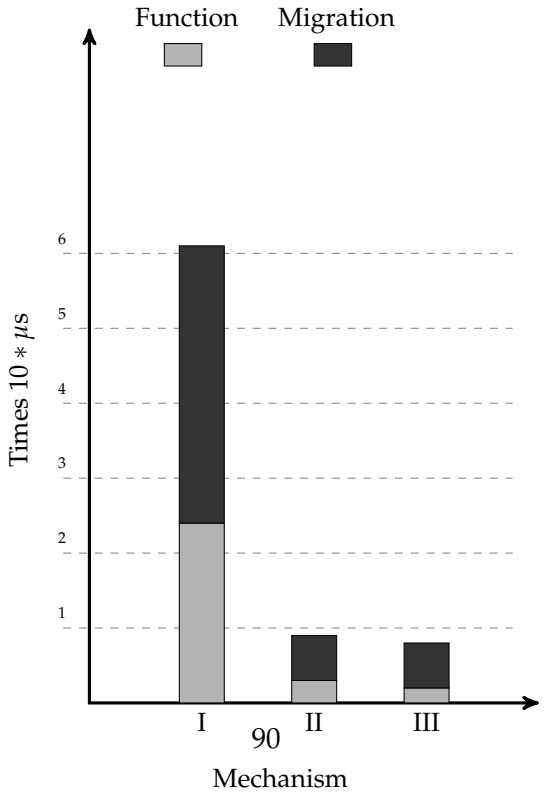
Alla luce di queste considerazioni, come accennato in precedenza, MrsP risulta utile in quelle situazioni in cui la somma degli overhead hanno un valore inferiore rispetto all'interferenza che viene causata dai job a priorità superiore, di conseguenza dipende dalla lunghezza della sezione critica. Se tali costi fossero superiori è conveniente un approccio basato sull'inibizione del prerilascio in quanto gli overhead pagati dal sistema sarebbero superiori ai benefici tratti dai job a priorità più alta del ceiling. La tabella 3.9 riassume i costi: il funzionamento di base ha un costo totale di circa 1300 ns (lock e release della risorsa), che quindi indica un tempo sotto il quale non è conveniente l'utilizzo del protocollo; nel caso in cui entrino in gioco i meccanismi descritti (per esempio, operazioni sulle CPU e migrazioni), il costo del protocollo aumenta, ma non è possibile



(a) Lock: overhead.



(b) Release: overhead.



(c) Finish-switch: overhead.

Figura 3.12: Overhead di sistema delle primitive.

Lock	800 ns	operazioni base
	2k ns	operazioni sulle CPU
	6k ns	migrazione del job
Release	500 ns	operazioni base
	2k ns	operazioni sulle CPU
	6k ns	migrazione del prossimo *lock holder*
	65k ns	migrazione del job
Finish-switch	24k ns	operazioni sullo stato
	37k ns	migrazione
	6k ns	notifica e migrazione *lock holder*

Tabella 3.9: Riassutno overhead di sistema delle primitive.

calcolare valore preciso in quanto non abbiamo controllo sulle esecuzioni dei job e, quindi, sul numero di migrazioni necessarie.

3.4 Impatto in assenza di risorsa globale

In un sistema real-time la condivisione di risorse globali tra task allocati su differenti CPU è un caso particolare, di conseguenza risulta interessante un confronto tra l'implementazione del protocollo con uno scheduler che non le prevede nell'esecuzione di un taskset in cui i task sono indipendenti.

3.4.1 Esperimento #3

Gli esperimenti discussi in questa sezione pongono a confronto l'implementazione di partenza di *partitioned fixed priority* fornita da LITMUS^{RT} con la versione modificata per integrare MrsP.

3.4.2 Configurazione

I taskset sono generati casualmente con alcuni accorgimenti: i periodi sono armonici e variano tra i 25ms ed i 200ms, i tempi di esecuzione sono calcolati con distribuzione di probabilità uniforme tra 0.1 e 0.4 in relazione al periodo, l'esecuzione dell'ultimo task creato viene ridimensionata in base alle esigenze per ottenere un utilizzazione dell'intero taskset pari al valore richiesto. Il problema del *bin-packing* per allocare i task nei processori segue l'euristica *worst-fit*. Infine, ogni esecuzione osservata ha una durata di 15 secondi.

Il valore di utilizzazione del taskset è pari alla somma del rapporto tra WCET e periodo di ogni task ed il carico di lavoro di sistema. L'esperimento prevede che i taskset siano creati in modo tale da rispettare una soglia di utilizzazione che varia dal 50% del carico totale fino al 100%, nel caso di un sistema composto da 4 processori le soglie saranno 2.0, 2.4, 2.8, 3.0, 3.2, 3.4, 3.6, 4.0.

3.4.3 Obiettivo

L'obiettivo è studiare le esecuzioni dei taskset e il comportamento degli scheduler all'aumentare del fattore di utilizzazione. Quello che ci si aspetta è che l'overhead aggiunto dall'integrazione di MrsP non vada ad intaccare il normale svolgimento di *partitioned fixed priority*, ma ad alti livelli di carico del sistema anche i minimi costi possono causare deadline miss che altrimenti non avverrebbero, questo considerando che già il sistema di per sé aggiunge dei ritardi dati dalle singole primitive.

La fase di valutazione rileva un aumento nel numero di *deadline miss* a partire da un rapporto di utilizzazione vicino al 70%, abbiamo quindi aumentato la frequenza di campionamento per carichi vicini all'80%. L'obiettivo è di evidenziare un valore di utilizzazione in cui alcuni taskset schedulabili, o con un limitato numero di *deadlin miss*, non lo siano nella versione che prevede task non indipendenti.

3.4.4 Risultati esperimento #3

Dalle esecuzioni sono stati estrapolati diverse informazioni: il numero di deadline miss rappresentato nel grafico 3.13 indica come il comportamento delle due implementazioni sia il medesimo. Nel grafico non sono riportati i dati relativi a workload pari al 90% ed al 100% in quanto già al 90% sei taskset su dieci producono deadline miss su entrambi; essendo i periodi armonici, a ogni iper-periodo le stesse dinamiche sono ripetute fino alla fine dell'esecuzione, pertanto ci si aspetta un numero molto alto. Di conseguenza, gli overhead degli scheduler non influiscono sui dati rilevati.

Le primitive principali utilizzate dal protocollo di accesso sono state discusse in precedenza; l'implementazione di *fixed priority scheduler* in assenza di risorsa condivisa non richiede le operazioni di accesso e rilascio, ergo sono state prese in considerazione l'evento di dispatching, di *context switch* e di *job release*.

L'operazione di dispatching ha il compito di decidere a quale job assegnare l'esecuzione nel processore, consiste quindi nel determinare lo stato del job

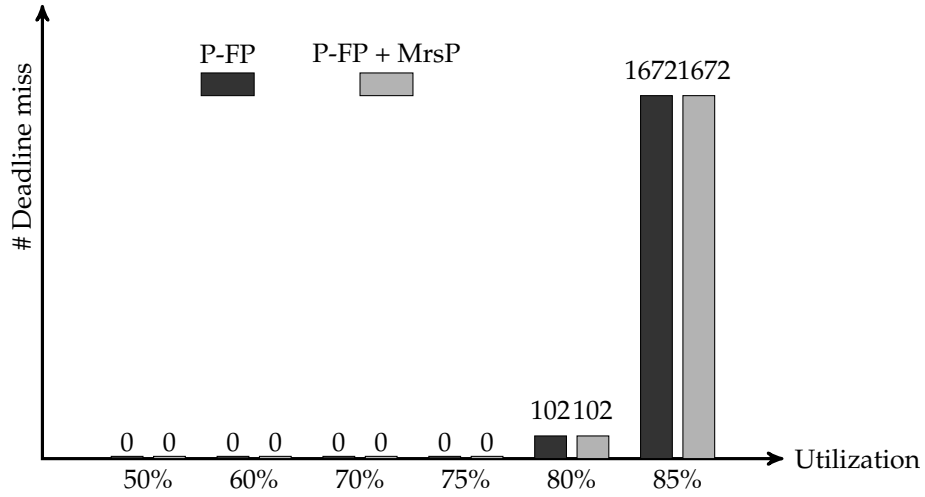
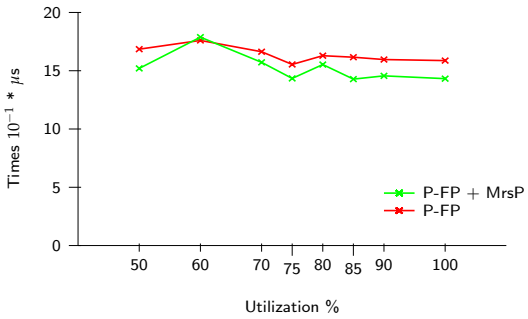


Figura 3.13: Numero di *deadline miss*.

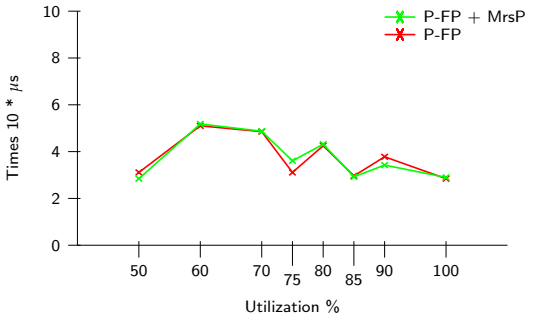
corrente e se necessario prerilasciarlo a favore di quello in testa alla coda dei job pronti all'esecuzione. Tale primitiva è richiamata in molteplici circostanze e il job in coda acquisisce la CPU quando è inutilizzata o ha priorità superiore rispetto a quello in esecuzione. Le figure 3.14a, 3.14b e 3.14c rappresentano e confrontano rispettivamente costo minimo, massimo e medio della primitiva di dispatching. In tutti e tre i grafici si nota come le due implementazioni abbiano lo stesso comportamento e la differenza tra i valori ottenuti è bassa, soprattutto per quanto riguarda il costo massimo.

Nei grafici 3.15a, 3.15b e 3.15c i rilevamenti dell'operazione di release mostrano nuovamente un andamento molto simile tra i due scheduler anche se con valori leggermente differenti. Tali divergenze non sono comunque riportabili all'implementazione in quando tale primitiva non viene modificata, inoltre le normali dinamiche di *partitioned fixed priority* non sono alterate. All'evento di release lo scheduler ha il compito di inserire il job nella coda ready, in questo caso mantenendo l'ordinamento in base alla priorità.

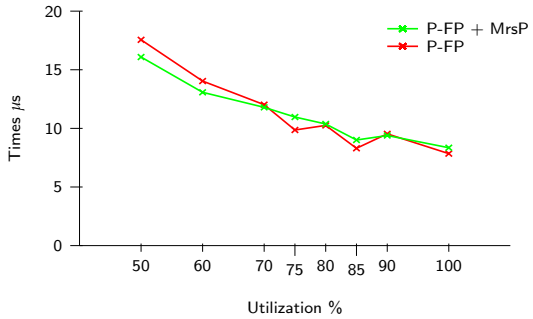
L'operazione di post-schedule è fondamentale nell'integrazione di MrsP, in essa vengono attuati meccanismi che permettono al job che detiene la risorsa di migrare quando necessario, al contrario in assenza di risorsa essa non ha alcun particolare utilizzo. Il costo minimo, massimo e medio risulta in linea con l'implementazione che non prevede condivisione di risorse: figure 3.16a, 3.16b e 3.16c.



(a) Costo minimo.



(b) Costo massimo.



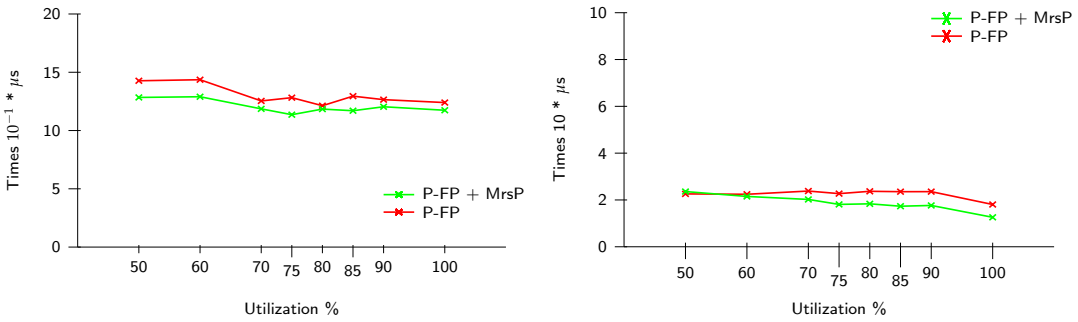
(c) Costo medio.

Figura 3.14: Confronto tra implementazioni: *schedule*.

3.4.5 Considerazioni

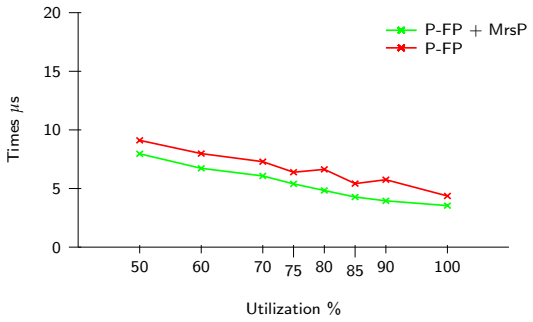
L'esperimento ha lo scopo di mettere a confronto il medesimo *scheduler* con e senza l'integrazione con *MrsP*.

I dati raccolti mostrano come nel caso di assenza di risorse globali non vi sia differenza tra le due, questo è un risultato importante in quanto la condivisione di risorse tra job allocati in diversi processori è un caso particolare e non quello normale. Di conseguenza, possiamo affermare che gli overhead del protocollo di accesso non hanno un impatto sul sistema quando non viene utilizzato.



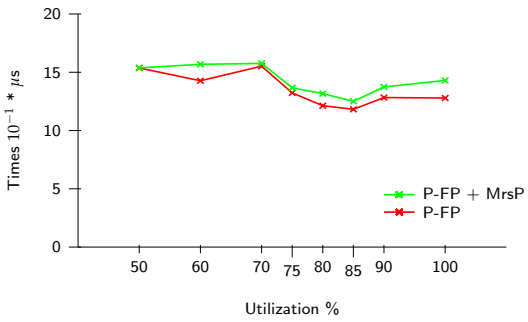
(a) Costo minimo.

(b) Costo massimo.

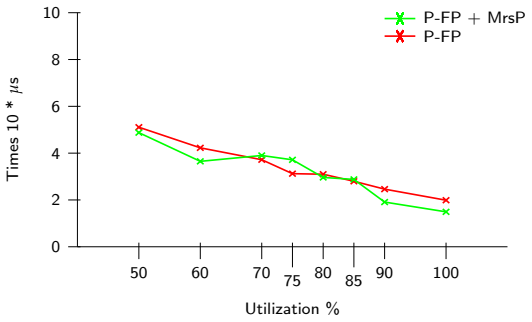


(c) Costo medio.

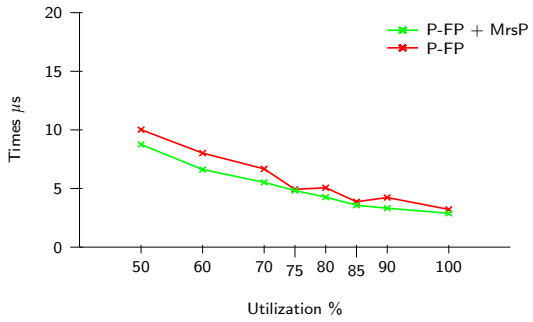
Figura 3.15: Confronto tra implementazioni: *job release*.



(a) Costo minimo.



(b) Costo massimo.



(c) Costo medio.

Figura 3.16: Confronto tra implementazioni: *context switch*.

Capitolo 4

Conclusioni

Questa tesi presenta l'implementazione e la valutazione empirica di un protocollo di accesso a risorsa disegnato a partire dalle linee guida fornite da Burns e Wellings in *A Schedulability Compatible Multiprocessor Resource Sharing Protocol - MrsP* ([9]). I due autori hanno dimostrato come il protocollo permetta l'utilizzo di tecniche di analisi di schedulabilità per sistemi single processor. In particolare, è presa in considerazione la *response-time analysis* che bene si presta per l'analisi di un sistema basato su scheduler *(Partitioned) Fixed Priority* aumentata dalla contesa per l'accesso alla risorsa globale. Di conseguenza, una volta calcolato il possibile numero di accessi in parallelo, questo parametro è utilizzato, in fase di analisi, per limitare l'impatto della loro serializzazione.

MrsP è un algoritmo basato su attesa attiva, cioè l'attesa effettuata dal job dopo la sua richiesta nel caso di risorsa occupata. In questo caso, infatti, la richiesta viene inserita in una coda con ordinamento FIFO fino a che non ottiene ed esegue la risorsa. Tale coda ha lunghezza massima pari al numero di processori in cui vi è allocato un task che la richiede. Come si è visto, la caratteristica che rende il protocollo innovativo consiste nel permettere a un qualsiasi job in attesa di accedere alla risorsa e di eseguire per conto del suo possessore. Nell'implementazione fornita, questo si traduce nel permettere al job, possessore della risorsa, di migrare, in caso di prerilascio, nella partizione di uno dei job in attesa.
MrsP definisce un protocollo con un basso impatto sulla schedulabilità di un taskset: la soluzione proposta è stata implementata per verificare se il protocollo comporta un alto costo in fase di esecuzione, andando quindi a penalizzare il basso impatto ottenuto dalle scelte algoritmiche.
‘ Gli esperimenti sono stati realizzati per valutare il protocollo e la relativa implementazione da diversi punti di vista. Attraverso la comparazione con

altri due protocolli: *simple ceiling* e *non preemption*, basati rispettivamente su utilizzo di ceiling e inibizione di prerilascio, i cui approcci sono alla base di molti dei protocolli precedentemente esistenti per la condivisione di risorse in sistemi multiprocessor. Dal confronto è emerso come MrsP garantisca un tempo di risposta inferiore rispetto agli altri due protocolli, sia per i job a bassa priorità che a priorità alta.
Inoltre, sono stati valutati i costi delle singoli primitive implementate, evidenziando così un limite entro il quale l'utilizzo del protocollo comporta beneficio; superato quel valore, invece, il costo aggiunto da MrsP diviene superiore rispetto alla durata della sezione critica della risorsa. I costi rilevati hanno dimostrato che le operazioni sulla coda, struttura su cui si basa il funzionamento del protocollo, comportano un costo ridotto rispetto al tempo complessivo necessario per l'esecuzione delle primitive di scheduling e di gestione della risorsa. Per esempio, la primitiva di acquisizione della risorsa ha un costo (nel caso pessimo) di circa 3k ns e solamente 200 ns circa sono riportabili alle operazioni sulla coda.
Infine, si è verificato come il protocollo non vada ad appesantire il sistema quando non vi sono risorse globali. Questo è un risultato importante in quanto, in un sistema reale, un task necessita della risorsa per un tempo breve rispetto alla lunghezza dell'esecuzione complessiva.

4.1 Sviluppi futuri

Limitazione delle migrazioni Gli esperimenti eseguiti hanno evidenziato come il costo principale del protocollo sia la migrazione. Un studio più approfondito potrebbe portare a un approccio in grado di limitarne il numero durante l'esecuzione. Una volta stimato il costo del protocollo su una determinata piattaforma e confrontato con la durata della sezione critica delle risorse, un approccio simile a FMLP (sezione 1.7.3) permette di dividerle tra brevi e lunghe. Per le prime si potrebbe optare per una gestione che inibisce il prerilascio, in quanto, come discusso, comporta un basso costo durante l'esecuzione. MrsP sarebbe chiamata a gestire gli accessi alle risorse lunghe, ma con un ulteriore accorgimento: quando un job a priorità superiore al ceiling richiede di prerilasciare il job in possesso della risorsa, si valuta di quanto tempo necessita quest'ultimo per portare a termine la sezione critica, se tale tempo è inferiore al tempo necessario per effettuare una migrazione viene ritardato il prerilascio, altrimenti vengono utilizzati i normali meccanismi del protocollo. Un approccio di questo tipo causerebbe blocco ai job a priorità superiore al ceiling, ma per un tempo limitato (minore al costo di una migrazione), per una

volta soltanto e prima dell'inizio dell'effettiva esecuzione. Di conseguenza, è facilmente integrabile in fase di analisi di schedulabilità.

Confronto con OMIP La sezione 1.7 ha analizzato il protocollo OMIP ([4]), il quale rappresenta una valida alternativa a MrsP. Nonostante sia pensato per sistemi organizzati a cluster, se si pone la dimensione di ognuno pari a uno si ottiene un sistema partizionato, quindi è possibile e interessante un confronto tra MrsP e OMIP. Inoltre, è possible modificare OMIP per meglio adattarlo a un sistema partizionato, tuttociò sostituendo la coda FIFO per ogni processore (sezione 1.8) con un qualche meccanismo che vada a implementare il concetto di *token* da acquisire in ogni cluster.

Risorse innestate Un possibile sviluppo futuro, per quanto riguardo il protocollo, è il supporto alle risorse innestate, cioè l'acquisizione di una risorsa quando si è già in possesso di un'altra.
La soluzione più semplice è quella di vincolare le risorse a un ordinamento, deciso in fase di progettazione, secondo il quale una risorsa può accedere solamente quelle che la precedono. In tal modo, si evitano le situazioni di deadlock a scapito di una minore espressività progettuale.
FMLP propone l'utilizzo dei Group locks, già discussi nella sezione 1.7.3. Le risorse innestate vengono raggruppate in unico gruppo e per accedere a una risorsa bisogna acquisire il gruppo a cui appartiene, nonostante non necessiti di tutte le risorse. In questo caso, si previene il deadlock ma a scapito del parallelismo.
Infine, Ward et. al [21] propongo il protocollo RNLP (*real-time nested locking protocol*). È basato su un sistema a k-lock, con il quale limita il numero di richieste di accesso alle risorse che possono essere in attesa in un determianto momento, e su un meccanismo che utilizza i timestamp delle richieste per ottenere un protocollo che permette risorse innestate a grana fine.

Appendices

Appendice A

LITMUS^{RT}

Di seguito viene fornita una panoramica di approfondimento del plug-in LITMUS^{RT} e del suo funzionamento.

LITMUS^{RT} è una patch per aggiungere ad un kernel Linux un'estensione real-time, in particolare per creare uno scheduler multiprocessore real-time con supporto per la sincronizzazione. Il kernel viene modificato per supportare un modello di task periodici e plug-in modulari per creare scheduler. Esso fornisce supporto, nonché diverse implementazioni, per scheduler globali, partizionati e cluster.

Il suo obiettivo principale è quello di fornire una piattaforma di sviluppo e sperimentazione nell'ambito dei sistemi real-time. Esso mette a disposizione una serie di funzioni che rispondono a determinati eventi per creare un prototipo di scheduler. Tali funzioni sono elencate e descritte nella tabella A.1. Per coerenza con la piattaforma in questione in alcuni casi al posto del termine job o processo viene utilizzato task.

Tutte le funzioni sono opzionali, salvo `schedule()` e `complete_job()` per le quali è obbligatorio fornire un'implementazione, negli altri casi viene fornita una versione di default in caso di mancato utilizzo.

Le funzioni principali sono `schedule()`, `tick()` e `finish_switch()`, esse rispondo ad eventi di scheduling:

- `schedule()` viene richiamata quando il sistema necessita di selezionare il task da eseguire, a livello logico consiste in un *context switch* e viene eseguita nel processore in cui è stata richiesta;

- `tick()` viene richiamata ogni quanto di tempo e se necessario richiama la funzione di `schedule()`, è utile per quei scheduler basati su quanti di tempo, per esempio PFair;
- `finish_switch()` viene richiamata dopo ogni *context switch*, utile se lo scheduler prevede un particolare comportamento da parte del task che interrompe la propria esecuzione.

Un secondo insieme di funzioni mira a gestire gli eventi che caratterizzano il ciclo di esecuzione di un processo:

- `release_at()` notifica che il task sarà soggetto ad una serie di release, utile nel caso lo si debba modificare prima della effettiva esecuzione;
- `task_block()` notifica quando un task viene sospeso, quindi tolto dalla coda ready;
- `task_wakeup()` richiamata quando un task riprende ad eseguire, deve essere reinserito nella coda ready;
- `complete_job()` notifica il completamento del processo, deve essere aggiunto il prossimo processo alla coda release del processore.

Le altre funzioni messe a disposizione dall'interfaccia riguardano la fase di inizializzazione e finalizzazione dello scheduler e dei task.

Oltre all'interfaccia per la gestione dello scheduler LITMUS^{RT} dispone una serie di funzioni per la gestione delle risorse condivise. Tramite la struttura `litmus_lock_ops` si accede ad un insieme di funzioni che permettono di implementare le operazioni logiche alla base di una risorsa, tabella A.2.

La funzione `open()` viene richiamata in fase di inizializzazione da ogni task che a run-time ne richieda l'accesso. Alla prima esecuzione della funzione viene creata la risorsa ed inizializzata, questo prevede anche di aggiungere la risorsa, e relativo identificatore, ad un file condiviso (rtspin-locks)gestito da LITMUS^{RT} sul quale viene tenuto traccia delle risorse utilizzate dall'intero taskset. Alle chiamate successive di `open()` il sistema riconosce, tramite il file rtspin-locks, che la risorsa è stata precedentemente creata, quindi ne recupera un riferimento e lo mette a disposizione del nuovo task per eventuali modifiche. Tale funzione è utile nei casi in cui si voglia calcolare un ceiling di risorsa, ad ogni chiamata il valore viene aggiornato in base alla priorità del task chiamante. La funzione

Funzione	Evento
`schedule()`	Seleziona il task da eseguire
`tick()`	Richiama lo scheduler ogni quanto di tempo
`finish_switch()`	Opera in seguito ad un context switch
`release_at()`	Prepara il task ai successivi rilasci
`task_block()`	Toglie il task dalla coda ready
`task_wakeup()`	Aggiunge il task alla coda ready
`complete_job()`	Prepara il job al successivo release
`admit_task()`	Controlla che il nuovo task sia correttamente configurato
`task_new()`	Alloca ed inizializza lo stato del nuovo task
`task_exit()`	Dealloca lo stato del task
`activate_plugin()`	Alloca ed inizializza lo stato del plug-in
`deactivate_plugin()`	Dealloca lo stato del plug-in
`allocate_lock()`	Alloca un nuovo lock

Tabella A.1: Funzioni dell'interfaccia di LITMUS^{RT}

Funzione	Evento
`open()`	Creazione ed inizializzazione della risorsa
`close()`	Chiusura della risorsa, se necessario forza il suo rilascio
`lock()`	Gestione della richiesta di accesso alla risorsa
`unlock()`	Gestione del rilascio della risorsa
`deallocate()`	Dealloca la memoria occupata dalla risorsa

Tabella A.2: Funzioni dell'interfaccia per le risorse.

`close()` viene richiamata anch'essa in fase di terminazione da ogni task che ne ha richiesto l'accesso, se il task chiamante è sta eseguendo la sezione critica ne forza il rilascio.

Le funzioni `lock()` e `unlock()` permettono di gestire la richiesta di accesso alla risorsa ed il relativo rilascio. Il loro utilizzo cambia in base all'approccio previsto dal protocollo: in caso di *spin-based protocol* nella funzione `lock()` viene predisposta l'attesa attiva fino al turno del task che ha effettuato la richiesta, mentre per una gestione basata su sospensione il task viene aggiunto ad una coda e risvegliato al momento opportuno. La funzione di `unlock()` viene richiamata al momento del rilascio, in tal caso il suo comportamento deve rispecchiare la logica alla base del protocollo che si vuole implementare.

Appendice B

Librerie user-space

In [capitolo di introduzione] è stata discussa l'architettura di LITMUS^{RT}, questa sezione si sofferma sulla parte di interfacciamento che il sistema mette a disposizione per la creazione di strumenti user-space con l'obiettivo di rendere possibile l'utilizzo dello scheduler. In particolare una serie di system call e meccanismi che la libreria *liblitmus* utilizza per rendere più semplice l'esecuzione ed il controllo dei taskset.

Le system call possono essere raggruppate secondo le funzioni svolte:

- definizione di task: permette di creare processi come task real-time di LITMUS^{RT}, i parametri (per esempio, *WCET*, periodo, etc.) vengono riportati in una struttura interna al task definita `rt_task`;
- controllo dei job: controllare il numero di sequenza di un job, attendere il suo rilascio e notifica l'evento di completamento;
- system call per la misurazione degli overhead;
- creazione, lock ed unlock di semafori real-time: una risorsa è rappresentata da un oggetto a livello kernel che ne contiene le informazioni; i task che la accedono devono ottenere un riferimento al medesimo oggetto, ma per questioni di controllo e sicurezza il kernel deve attuare meccanismi che mirano a modificare il namespace della risorsa, in questo modo non permette un accesso diretto ai processi. Nelle versioni recenti di LITMUS^{RT} questo problema viene risolto tramite l'utilizzo di *i-node*, cioè la rappresentazione a livello kernel di un file presente nel *file system*: due task che condividono la medesima risorsa richiedono tramite system call accesso allo stesso file. Gli indirizzi dei file, definiti *file-descriptor-attached shared objects* (FDSOs), sono

salvati in una tabella di look up in modo tale da ridurre gli overhead dati dal kernel;

- sincronizzazione della release dell'intero taskset: questo permette di rilasciare al medesimo istante il primo job di ogni task. Questo avviene tramite una particolare chiamata di sistema, `wait_for_ts_release()`: i task vengono accodati su di una barriera fino a che il sistema non riconosce che tutti siano accodati e pronti ad eseguire, dopodiché la barriera viene aperta per ottenere un unico release. Tale evento viene rilevato da `release_at()`.

Per esportare informazioni riguardo all'esecuzione LITMUS^{RT} aggiunge quattro particolari dispositivi virtuali per permettere di controllare lo stato. Tre di questi hanno come obiettivo il tracciamento: `TRACE()` per il debug, `feather-trace` per il campionamento e `sched_trace()` per rilevare gli eventi di scheduling. Le tecniche di tracciamento vengono discusse in C. Il quarto device introduce meccanismi di *control page* per supportare *non-preemptive sections* con bassi overhead.

Entrare ed uscire da una sezione critica disabilitando gli *interrupt* richiede due *system call*, di conseguenza risulta essere un'operazione onerosa ed in alcuni casi gli overhead possono essere superiori alla lunghezza della stessa sezione critica. Il meccanismo di *control page* abilita il kernel ed ogni processo real-time alla condivisione di flags ed informazioni senza dover effettuare *system call*: quando un processo utilizza il dispositivo viene allocata una pagina di memoria del kernel e la mappa nello spazio degli indirizzi del processo. Per maggiori dettagli vedi [6].

La libreria liblitmus permette di utilizzare parte dell'interfaccia user-space messa a disposizione da LITMUS^{RT}. Nella tabella B.1 sono elencate e brevemente descritte le principali funzionalità.

Funzionalità	Scopo
`setsched`	Seleziona ed attiva un plugin.
`showsched`	Stampa il nome dello scheduler attualmente utilizzato.
`rt_launch`	Lancia un semplice processo come un task real-time specificando *WCET* e periodo.
`rtspin`	Semplice task utile per simulare l'utilizzo della CPU. Effettua cicli senza effettuare particolari operazioni per una durata basata sui parametri specificati. Se previste, effettua le chiamate per ottenere l'accesso alle risorse condivise, una volta ottenuta entra in loop fino allo scadere del tempo della sezione critica. E' possibile modificarlo per ottenere differenti comportamenti.
`release_ts`	Rilascia i task del sistema, utile per ottenere un'unica release sincronizzata per l'intero taskset.
`measure_syscall`	Strumento per la misurazione degli overhead della *system call*.
`cycles`	Mostra i cicli per un determinato intervallo di tempo.
`base_task`	Esempio di task real-time. Base per lo sviluppo di task *single-thread*.
`base_mt_task`	Esempio di task *multi-thread* real-time. Base per lo sviluppo di task *multi-thread*.

Tabella B.1: Strumenti user-space di *liblitmus*.

Appendice C

TRACE() e *Feather-Trace*

LITMUS^{RT} mette a disposizione due differenti tipi di tracciamento.

C.1 TRACE()

`TRACE()` fa parte dell'architettura del sistema e permette di effettuare debugging. Al pari di `printk()` esporta informazioni dal kernel. La necessità di introdurre una nuova macro è dettata dal fatto che `printk()` porta a deadlock, per approfondimenti [6].

Per evitare questo problema un dispositivo viene continuamente interrogato per esportare i messaggi di tracciamento. Questa soluzione risulta molto onerosa, di conseguenza una volta terminata la fase di debugging viene disabilitato.

C.2 *Feather-Trace*

Feather-Trace è un toolkit di tracciamento per sistemi che comporta bassi overhead. Esso permette di tracciare il comportamento del sistema raccogliendo informazioni riguardo allo stato ed alle performance durante l'esecuzione per poi analizzare tali dati offline. Questi strumenti sono stati scelti per la loro semplicità ed efficienza.

Mette a disposizione due componenti:

- *static trigger*, modifica il normale flusso tramite l'invocazione di callback fornite dall'utente, permettendo di effettuare le azioni volute. L'overhead di questa operazione è reso minimo dall'utilizzo dell'operazione assembly di *jump*.
- un buffer che permette scritture in ordinamento FIFO *multi-writer* senza attesa, utilizzato per raccogliere i dati degli overhead. Il processo che intende scrivere nel buffer non deve attendere e le operazioni di copia sono ridotte al minimo. Un processo non real-time periodicamente legge il buffer, lo svuota e ne trasferisce i dati in memoria.

Dettagli dell'implementazione possono essere trovati in [7].

In LITMUS^{RT} Feather-Trace viene utilizzato sia per il rilevamento dei eventi di scheduling sia per registrare timestamp per i campionamenti. Registrando due timestamp ad inizio e fine di una determinata operazione si è in grano di calcolare il costo di una primitiva o di un blocco di operazioni. Gli eventi sono elaborati offline per creare grafici per avere una visualizzazione dell'esecuzione del taskset, molto utile in base di debugging per trovare gli errori. Gli strumenti per l'elaborazione dei dati sono `sched_trace()` per gli eventi e `feather-trace` per il campionamento degli overhead.

Appendice D

experiment-scripts

Una breve panoramica su questo insieme di script entra nel dettaglio della fase di generazione ed esecuzione degli esperimenti, in particolare nel formato dei file necessari. *Experiment-scripts* fornisce un insieme di script python per facilitare la creazione ed esecuzione di un taskset ed infine l'analisi dei dati raccolti e la loro rappresentazione grafica.

- `gen_exps.py` per la creazione degli esperimenti;
- `run_exps.py` per l'esecuzione dell'esperimento;
- `parse_exps.py` per elaborare i dati del tracciamento di LITMUS^{RT};
- `plot_exps.py` per la rappresentazione grafica dei dati in formato csv.

`gen_exps.py` permette di creare uno o più esperimenti grazie ad un unico comando. Tramite alcuni parametri è possibile specificare per esempio il plugin scelto, il numero di task ed il numero di CPU.

L'esecuzione dello script genera due file contenuti in un unica folder il cui nome riporta i parametri principali specificati:

- `params.py` fornisce informazioni riguardanti l'esperimento, quindi lo scheduler scelto, il numero di CPU, se i periodi sono armonici, la distribuzione di utilizzazione tra i task, la durata dell'esecuzione del taskset, etc. Di seguito un esempio di configurazione del file:

```
1    'release_master': False,
2    'duration': 30,
3    'utils': 'uni-medium',
```

```
4     'scheduler': 'P-FP',
5     'cpus': 4
```

- `sched.py` contiene la lista di task che costituiscono l'esperimento secondo un formato ben preciso:

```
1 -p 0 -q 40 2 17.5
2 -p 1 -q 10 1 8.0
3 -p 2 -q 10 3 7.0
4 -p 2 -q 20 2 8.0
5 -p 2 -q 30 1 32.0
6 -p 3 -q 30 1 11.0
```

  Nell'esempio riportato sono presenti 7 task allocati su 4 CPU, per ognuno è specificata la partizione (`-p`), la priorità (`-q`), WCET e periodo. Questi parametri sono necessari per avviare l'esecuzione dei task tramite l'utilizzo di `rt-spin` (vedi B).

Lo script di generazione è stato modificato per poter specificare il rapporto di utilizzazione voluto dell'interno taskset generato, in modo tale da poterlo variare ed ottenere così esperimenti con workload crescenti. Un'ulteriore modifica è stata apportata per creare taskset che prevedano l'utilizzo di risorse. L'assegnazione della risorse ai task viene effettuata in modo casuale, ai task designati viene specificato il protocollo da utilizzare per accedere la risorsa, la lunghezza della sezione critica ed un identificativo della risorsa stessa. Il file `sched.py` dopo tale modifica risulta il seguente:

```
7  -p 0 -q 40 -X MRSP -L 0.5 -Q 1 2 17.5
8  -p 1 -q 10 1 8.0
9  -p 2 -q 10 3 7.0
10 -p 2 -q 20 -X MRSP -L 0.5 -Q 1 2 8.0
11 -p 2 -q 30 1 32.0
12 -p 3 -q 30 -X MRSP -L 0.5 -Q 1 1 11.0
```

Il nuovo taskset prevede che 3 task allocati nelle CPU 0, 2 e 3 condividano la risorsa con id 1 (`-Q 1`) gestita dal protocollo d'accesso MrsP (`-X MRSP`) ed infine che la sessione critica sia pari a mezzo millisecondo (`-L 0.5`).
`run_exps.py` richiede in input una cartella contenente i file `sched.py` e `params.py` ed utilizza la libreria *liblitmus* per attivare lo scheduler scelto, avviare i meccanismi di tracciamento, rilasciare i task ed infine raccoglie i dati riguardanti l'esecuzione, cioè gli overhead e gli eventi. Questi dati sono utili per l'elabo-

razione offline da parte di `sched_trace` e `feather-trace`.

`parse_exps.py` richiede in input i file di tracciamento generati dallo script precedente e ne genera delle statistiche riguardanti i singoli eventi ed overhead, infine `plot_exps.py` si appoggia alla libreria *Matplotlib* per generare grafici con i dati elaborati.

Bibliografia

1. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

2. T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, April 1991.

3. Aaron Block, Hennadiy Leontyev, Bjorn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56, Washington, DC, USA, 2007. IEEE Computer Society.

4. B.B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 292–302, July 2013.

5. Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 342–353, Washington, DC, USA, 2008. IEEE Computer Society.

6. Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

7. Björn B. Brandenburg and James H. Feather-trace: A light-weight event tracing toolkit. In *In Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications (OSPERT'07*, pages 61–70, 2007.

8. A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – mrsp. In *Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems*, ECRTS '13, pages 282–291, Washington, DC, USA, 2013. IEEE Computer Society.

9. Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.

10. Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

11. D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 90–99, July 2010.

12. Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '03, pages 189–, Washington, DC, USA, 2003. IEEE Computer Society.

13. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

14. K. Lakshmanan, D. De Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 469–478, Dec 2009.

15. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

16. Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

17. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990.

18. Lui Sha, Tarek Abdelzaher, Karl-Erik AArzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Syst.*, 28(2-3):101–155, November 2004.

19. H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 134–143, Dec 1997.

20. Linus Torvalds. Spin locks. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/spinlocks.txt`.

21. Bryan C. Ward and James H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS*, pages 223–232, 2012.