PROGRAMMAZIONE DI SISTEMA

A.A. 2022/2023

S308786 OLIVA MATTIA

S319103 SEFA ENDRI


Politecnico di Torino

IMPLEMENTAZIONE DI VARI

# ALGORITMI DI SCHEDULING IN MENTOS

# INTRODUZIONE

- MentOS consente di selezionare l'algoritmo di scheduling tra varie opzioni:
  - Round Robin (RR)
  - Priority Scheduling (PS)
  - Completely Fair Scheduling (CFS)
  - Rate Monotonic (RM)
  - Earliest Deadline First (EDF)
  - Aperiodic EDF (AEDF)
- Di questi, nel codice di base di MentOS, solo il RR è effettivamente implementato, il secondo e il terzo sono quasi completi ma privi della parte fondamentale di selezione del prossimo task, mentre gli ultimi tre sono completamente vuoti (se no per la chiamata al RR nel caso in cui vengano erroneamente selezionati)

- I vari algoritmi sono selezionabili al momento della compilazione del kernel
  - Mediante il comando "`cmake -DSCHEDULER_TYPE=X ..`" dove X è l'opzione definita per l'algoritmo desiderato.
  - Accedendo al menù di compilazione tramite il comando "`ccmake ..`", selezionando l'algoritmo desiderato (opzione `SCHEDULER_TYPE`) e salvando le impostazioni
- L'impostazione selezionata verrà poi usata all'interno della *scheduler_pick_next_task* (sita in *src/process/scheduler_algotithm.c*) per selezionare la funzione corrispondente da invocare per la scelta della successiva task da eseguire

```
task_struct *scheduler_pick_next_task(runqueue_t *runqueue)
{
    // Update task statistics.
#if (defined(SCHEDULER_CFS) || defined(SCHEDULER_EDF) || defined(SCHEDULER_RM) || defined(SCHEDULER_AEDF) || defined(SCHEDULER_LLF))
    __update_task_statistics(runqueue->curr);
#endif

    // Pointer to the next task to schedule.
    task_struct *next = NULL;
#if defined(SCHEDULER_RR)
    next = __scheduler_rr(runqueue, false);
#elif defined(SCHEDULER_PRIORITY)
    next = __scheduler_priority(runqueue, false);
```

# REAL TIME SCHEDULING

- MentOS supporta il Real-Time scheduling (cioè che si concentra sull'esecuzione di task o processi entro determinati vincoli di tempo), pertanto la coda di esecuzione può contenere sia task periodiche sia aperiodiche
- RM, EDF e AEDF fanno tutti parte degli algoritmi di real-time scheduling
- In aggiunta ai tre precedenti, abbiamo deciso id implementare un ulteriore algoritmo di questo tipo, il *Least Laxity First (LLF)*

- Componente fondamentale del real-time scheduling è l'**analisi di schedulabilità**, che consiste nel verificare che l'aggiunta della task alla coda d'esecuzione non causi alle altre task di mancare le proprie deadline
- In MentOS il compito di effettuare quest'analisi è della funzione *sys_waitperiod()* (in *lib/process/scheduler.h*) che analizza la task in questione eseguendola la prima volta come se non fosse periodica e calcolandone il tempo massimo di esecuzione possibile (Worst Case Execution Time *WCET*)
- Noto il WCET, la waitperiod definisce se la funzione è schedulabile o meno facendo riferimento all'algoritmo scelto e agli altri processi in coda
- NOTA: la *sys_waitperiod()* è già implementataall'interno di MentOS base, quindi non verrà trattata all'interno di queste slides

# IMPLEMENTAZIONE

- Tutto il codice implementato di seguito si trova all'interno di *src/process/scheduler_algorithm.c* salvo diversamente specificato
- Per poter implementare/completare l'implementazione delle funzioni seguenti abbiamo dovuto, nell'ordine:
  1. Comprendere gli algoritmi in sé
  2. Analizzare e comprendere le strutture dati usate dallo scheduler messe a disposizione dai progettisti di MentOS poter operare su di esse. (È stata adoperata un'unica struttura dati, la *scheduler_entity_t*, contenente campi utili ad ogni algoritmo)
  3. Implementare gli algoritmi visti basandosi sulle strutture dati fornite e le funzioni di gestione di tali strutture
- In aggiunta agli algoritmi precedentemente elencati, abbiamo deciso di implementare il *Least Laxity First* (LLF)

# STRUTTURE DATI

- Di seguito riportiamo solo un elenco sommario delle strutture dati utilizzate:
  - *runqueue_t runqueue:* dichiarata staticamente all'interno di *src/process/scheduler.c,* è la coda di esecuzione delle task. Da questa lo scheduler deve selezionare la prossima task da eseguire. (Per una descrizione più dettagliata della struct si faccia riferimento alle slide "*MentOS – Process Management*")
  - *task_struct:* struct usata per rappresentare una task. Sono i nodi della runqueue
  - *scheduler_entity_t:* è la struttura dati che mantiene le informazioni utili alla schedulazione della singola task. Ogni *task_struct* presenta un campo (denominato "se") di questo tipo

```c
typedef struct sched_entity_t {
    /// Static execution priority.
    int prio;
    /// Start execution time.
    time_t start_runtime;
    /// Last context switch time.
    time_t exec_start;
    /// Last execution time.
    time_t exec_runtime;
    /// Overall execution time.
    time_t sum_exec_runtime;
    /// Weighted execution time.
    time_t vruntime;
    /// Expected period of the task
    time_t period;
    /// Absolute deadline
    time_t deadline;
    /// Absolute time of arrival of the task
    time_t arrivaltime;
    /// Has already executed
    bool_t executed;
    /// Determines if it is a periodic task.
    bool_t is_periodic;
    /// Determines if we need to analyze the WCET of the process.
    bool_t is_under_analysis;
    /// Beginning of next period
    time_t next_period;
    /// Worst case execution time
    time_t worst_case_exec;
    /// Processor utilization factor
    double utilization_factor;
} sched_entity_t;
```

- La struct sched_entity_t come appare in *lib/process/process.h*
- Senza dubbio la struttura dati più importante per l'implementazione svolta, in quanto contiene vari dati utili, tra cui:
  - tempo totale di esecuzione
  - deadline
  - periodicità
  - priorità
  - tempo di arrivo

# PRIORITY FIRST

- Come visibile nelle slide "*MentOS – Process Management*", il codice della funzione *__scheduler_priority*, che riceve come parametri la runqueue ed un booleano (indicante se tenere conto delle task periodiche), e restituisce la prossima task da eseguire, è pressoché completa nel codice di base di MentOS. A mancare sono solo la parte inerente alla scelta della task e la lettura della priorità della task corrente
- Per effettuare la scelta si va a cercare la task con il valore *se.prio* più basso (corrispondente ad un livello di priorità più alta, di fatto). A parità di priorità dei processi, si può optare per diverse strategie. Noi effettuiamo una scelta arbitraria, selezionando sempre la task con cui ci stiamo confrontando invece di quella attuale. Una strategia alternativa tra le molte potrebbe essere quella di basarsi sul tempo di arrivo (scegliendo poi se servire la task più nuova o più vecchia)

```
static inline task_struct *__scheduler_priority(runqueue_t *runqueue, bool_t skip_periodic)
{
#ifdef SCHEDULER_PRIORITY
    // Get the first element of the list.
    task_struct *next = list_entry(runqueue->queue.next, struct task_struct, run_list);

    // Get its static priority.
    time_t min = next->se.prio;
```

- La lettura della priorità della task in esame. Il dato è contenuto nel campo *prio* della struct *se*, a sua volta attributo della task

```
    // Check if the entry has a lower priority.
    if (entry->se.prio <= min) {
            next = entry;
            min = entry->se.prio;
    }
```

- La strategia adottata per la selezione delle task. La condizione '<=' assicura che in caso di parità prenderemo l'altro processo invece di quello corrente

- Per vedere il codice per intero (salvo queste parti mancanti nell'originale) si faccia riferimento alle slide sulla gestione dei processi precedentemente citate

# COMPLETELY FAIR SCHEDULING

- Similmente al PF, anche il CFS si trova quasi completamente implementato nel codice di MentOS di base. Alla funzione in sé mancavano le stesse parti di PF (salvo dover leggere il *vruntime* invece della priorità)
- L'algoritmo CFS prevede una funzione supplementare con il compito di aggiornare i pesi delle task (ovvero i vruntime). Anche questa funzione presentava già uno "scheletro", seppur meno completo della funzione di scheduling stessa
- La funzione di scheduling *__scheduler_cfs* prende come parametri i medesimi visti per la PF e restituisce sempre la nuova task scelta
- La funzione di aggiornamento dei pesi, *__update_task_statistics*, prende come parametro solo la task corrente, e viene invocata dalla *scheduler_pick_next_task* prima della chiamata alla funzione di scheduling

- L'implementazione della funzione in se è molto semplice, si tratta in patica sempre di scorrere la lista di task in attesa, saltando quelle periodiche a seconda del valore del parametro booleano passato alla funzione, e scegliere la prossima task come quella con *vruntime* minimo (invece di confrontarne la priorità). Il fatto che il *vruntime* cambi dinamicamente e indipendentemente rispetto agli altri ci assicura che una task non possa rimanere troppo a lungo in possesso della cpu

```
// Get its virtual runtime.
time_t min = next->se.vruntime;
//…
// Search for the task with the smallest vruntime value.
list_for_each_decl(it, &runqueue->curr->run_list)
{   //…
    // Check if the element in the list has a smaller vruntime value.
    if (entry->se.vruntime < min) {
        next = entry;
        min = entry->se.vruntime;
    }
```

```c
static void __update_task_statistics(task_struct *task)
{
    // See `prio.h` for more support functions.
    assert(task && "Current task is not valid.");

    // While periodic task is under analysis is executed with aperiodic
    // scheduler and can be preempted by a "true" periodic task.
    // We need to sum all the execution spots to calculate the WCET even
    // if is a more pessimistic evaluation.
    // Update the delta exec.
    task->se.exec_runtime = timer_get_ticks() - task->se.exec_start;

    // Perform timer-related checks.
    update_process_profiling_timer(task);

    // Set the sum_exec_runtime.
    task->se.sum_exec_runtime += task->se.exec_runtime;

    // If the task is not a periodic task we have to update the virtual runtime.
    if (!task->se.is_periodic) {
        // Get the weight of the current task.
        time_t weight = GET_WEIGHT(task->se.prio);
        // If the weight is different from the default load, compute it.
        if (weight != NICE_0_LOAD) {
            // Get the multiplicative factor for its delta_exec.
            double factor = ((double)NICE_0_LOAD)/((double)weight);
            // Weight the delta_exec with the multiplicative factor.
            task->se.exec_runtime = (int)(((double)task->se.exec_runtime)*factor);
        }
        // Update vruntime of the current task.
        task->se.vruntime += task->se.exec_runtime;
    }
```

Più interessante è l'implementazione della funzione di aggiornamento del *vruntime* di una task.

1. Per prima cosa si calcola il *delta_exec* (salvato nel campo *exec_runtime* della *se*) come differenza tra adesso ed il tempo d'inizio dell'esecuzione.
2. A questo punto si calcola il peso della task usando una tabella che correla priorità e peso, e in caso questo peso non corrispondesse al carico di default (NICE_0_LOAD) si scala il *delta_exec* della task.
3. A questo punto si aggiorna il *vruntime* della task aggiungendovi il *delta_exec*

# RATE MONOTONIC

- Il *rate monotonic* è un algoritmo basato su una priorità fissa, che viene assegnata ad ogni task al momento della sua analisi di schedulabilità.
- Maggiore la frequenza di richiesta (a.k.a. minore il periodo della task), maggiore sarà la priorità
- L'arrivo di una task con periodo minore comporta una priorità maggiore di quella delle task attuali. Di conseguenza il *rate monotonic* è intrinsecamente pre-emptive (a.k.a. la task attuale può essere interrotta per dare priorità ad un'altra)
- L'algoritmo in sé è molto semplice, si tratta di scegliere la task per cui il valore *se.next_period* sia minimo, escludendo quella appena eseguita. Se non vengono trovate task periodiche con i requisiti richiesti, si passa ad usare il *CFS*, potendo in questo modo gestire sia task periodiche sia aperiodiche nella stessa coda di esecuzione

```
static inline task_struct *__scheduler_rm(runqueue_t *runqueue)
{
    //pointer to the next task
    task_struct *next = NULL, *entry;
    //the next period, starting from the maximum possible one
    time t next np = UINT MAX;
    //iterate over the tasks list looking for the closest next period
    list_for_each_decl(it, &runqueue->queue){
        //if we're at the head, we skip it
        if(it == &runqueue->queue)
            continue;
        //gets the task_struct from the list node
        entry = list_entry(it, task_struct, run_list);
        //we skip non-period tasks or a periodic task that's still undergoing schedula
        if(!entry->se.is_periodic || entry->se.is_under_analysis)
            continue;
        if(entry->se.executed && (entry->se.next_period <= timer_get_ticks())){
            entry->se.executed = false;
            entry->se.deadline += entry->se.period;
            entry->se.next_period += entry->se.period;
        }//if it's not marked as executed I check if it's the closest deadline
        else if(!entry->se.executed && (entry->se.next_period < next_np)){
            next = entry;
            next_np = next->se.next_period;
        }
    }
    //then if i haven't found a valid periodic task, i use the CFS
    if(next == NULL)
        next = __scheduler_cfs(runqueue, true); //true = skips periodic tasks
    return next;
}
```

- L'algoritmo di base è prettamente simile ai precedenti: si itera sulla coda di esecuzione cercando le entry periodiche che non sono più sotto analisi.
- Una vota trovata una di queste entry, si verifica che non sia quella appena eseguita. In caso lo fosse, se ne aggiornano i parametri (incrementando la deadline e il prossimo periodo di 1 periodo, e settandone il flag *se.executed* a falso)
- Se la entry trovata non è quella appena eseguita, si controlla che sia quella con il *se.next_period* più vicino e in caso affermativo la si prende come next
- Se terminato il ciclo sulla queue non si ha ancora trovato un next, si procede con lo scheduling CFS

# EARLIEST DEADLINE FIRST

- Questo algoritmo si basa sul selezionare la task con la deadline più vicina
- Come il precedente, anche questo algoritmo consente di gestire automaticamente la pre-emption e l'arrivo dinamico di task nella coda d'esecuzione
- L'implementazione è pressoché identica al precedente, pertanto nella slide seguente ci limiteremo ad evidenziare le parti di codice che differiscono, ovvero quelle relative alla condizione di scelta della task successiva
- Anche in questo caso, se una volta terminate le iterazioni sulla coda non si è trovata nessuna task periodica che rispetti i criteri, si invoca la CFS

```
static inline task_struct *__scheduler_edf(runqueue_t *runqueue)
{
    //pointer to the next task
    task_struct *next = NULL, *entry;

    //the next deadline, starting from the maximum possible one
    time_t next_dl = UINT_MAX;
```

- Questa volta siamo alla ricerca della deadline più vicina (ovvero quella più piccola)

```
}//if it's not marked as executed I check if it's the closest deadline
else if(!entry->se.executed && (entry->se.deadline < next_dl)){
    next = entry;
    next_dl = next->se.deadline;
}
```

- Verifico che la task in esame non sia l'ultima eseguita e che abbia la deadline più prossima tra quelle trovate finora
- In caso affermativo, salvo la entry in next, aggiorno la deadline minima e procedo alla prossima entry della *runqueue*

# APERIODIC EARLIEST DEADLINE FIRST

- Questo algoritmo funziona allo stesso modo dell' EDF, ma non effettua l'analisi di schedulabilità
- Può quindi portare alcune task a mancare la propria deadline, come conseguenza di quanto detto sopra
- L'implementazione di fatto consiste in una versione "ridotta" di quanto visto per l'algoritmo EDF, in quanto si effettuano i controlli di periodicità né si aggiornano i periodi di eventuali task, che vengono trattate tutte come **aperiodiche**
- Rimane tuttavia la scelta basata sulla prossimità della deadline per la prossima task

# LEAST LAXITY FIRST

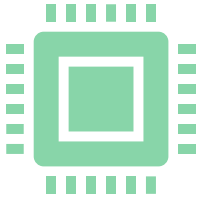
- È basato sull'assegnare priorità ai task in base al loro livello di "laxity" (slack)
- La laxity è calcolata come il tempo rimanente prima della sua scadenza meno il tempo di esecuzione rimanente. In altre parole, la laxity rappresenta quanto tempo un task può attendere prima di essere eseguito senza compromettere la sua scadenza
- L'obiettivo principale dell'algoritmo LLF è massimizzare il rispetto delle scadenze, poiché le task con scadenze più vicine e meno margine di tempo dovrebbero essere eseguite prima

- L'algoritmo rimane molto simile a quelli di CFS e RM, cambiando la condizione di selezione. Il procedimento quindi rimane:
  1. Ciclo la coda di esecuzione
  2. Ad ogni elemento, aggiorno flag e periodo se è l'ultima task eseguita, ed effettuo altri controlli (es: se è periodico)
  3. Una volta trovata una entry valida verifico che sia quella con laxity minima, in caso affermativo la salvo in *next* e aggiorno la laxity minima
  4. Una volta terminata la lista, restituisco l'entry se l'ho trovata, altrimenti invoco la CFS
- Al contrario di quelli visti in precedenza, per questo algoritmo non esiste nemmeno il supporto "scheletrico" nel codice di base di MentOS. Per poter effettuare la compilazione condizionale, è stato quindi necessario andare a modificare il file mentos/CMakeLists.txt ed inserire il valore SCHEDULER_LLF nella lista di opzioni per lo scheduler
- È stato inoltre necessario modificare la funzione *scheduler_pick_next_task* affinché richiamasse la *__scheduler_llf* e *__update_task_statistics* quando l'opzione è selezionata

```c
static inline task_struct *__scheduler_llf(runqueue_t *runqueue)
{
    //pointer to the next task
    task_struct *next = NULL, *entry;
    //the next period, starting from the maximum possible one
    time_t min_lax = UINT_MAX;
    //iterate over the tasks list looking for the closest next period
    list_for_each_decl(it, &runqueue->queue){
        //if we're at the head, we skip it
        if(it == &runqueue->queue)
            continue;
        //gets the task_struct from the list node
        entry = list_entry(it, task_struct, run_list);
        //we skip non-period tasks or a periodic task that's still undergoing schedulability analysis
        if(!entry->se.is_periodic || entry->se.is_under_analysis)
            continue;
        if(entry->se.executed && (entry->se.next_period <= timer_get_ticks())){
            entry->se.executed = false;
            entry->se.deadline += entry->se.period;
            entry->se.next_period += entry->se.period;
        }//if it's not marked as executed I check if it's the least laxed
        else if(!entry->se.executed){
            //get's how much time till the deadline
            //laxity = remaining time - total execution time up to now
            int this_lax = (entry->se.deadline - timer_get_ticks()) - entry->se.sum_exec_runtime;
            if(this_lax < min_lax){
                next = entry;
                min_lax = this_lax;
            }
        }
    }
    //then if i haven't found a valid real time task, i use the CFS
    if(next == NULL)
        next = __scheduler_cfs(runqueue, true); //true = skips periodic tasks
    return next;
}
```

- Questa volta siamo alla ricerca della laxity minima (*min_lax*)
- La laxity viene calcolata come il tempo rimanente alla deadline meno il tempo totale d'esecuzione finora passato
- I controlli sono i medesimi fatti per RM ed EDF:
  - Dev'essere periodico e aver terminato l'analisi
  - Se è l'ultimo ad essere stato eseguito lo si aggiorna e si va avanti
  - Se non è appena stato eseguito si controlla la laxity e se è la minima si salva la entry e si aggiorna *min_lax*

# SOURCES & CREDITS



## Sources:

MentOS: https://mentos-team.github.io/doc/doxygen/index.html

Linux Kernel: "Understanding the Linux Kernel, Third Edition 3rd Edition", M. Cesati, D. P. Bovet

## Credits and Thanks:

All of the previous

## Copyright Licence:

CONTATTI:

ENDRI.SEFA@STUDENTI.POLITO.IT

MATTIA.OLIVA@STUDENTI.POLITO.IT