PROCESS MANAGEMENT
E
ANALISI DEGLI ALGORITMI DI SCHEDULING
IN
MENTOS

PROGRAMMAZIONE DI SISTEMA

A.A. 2022/2023

S319103 SEFA ENDRI

S308786 OLIVA MATTIA

Politecnico di Torino

INDICE

- Processi e loro funzionamento in MentOS
- Scheduler
- Algoritmi di Scheduling
- Paragone con OS161

PROCESS DESCRIPTOR

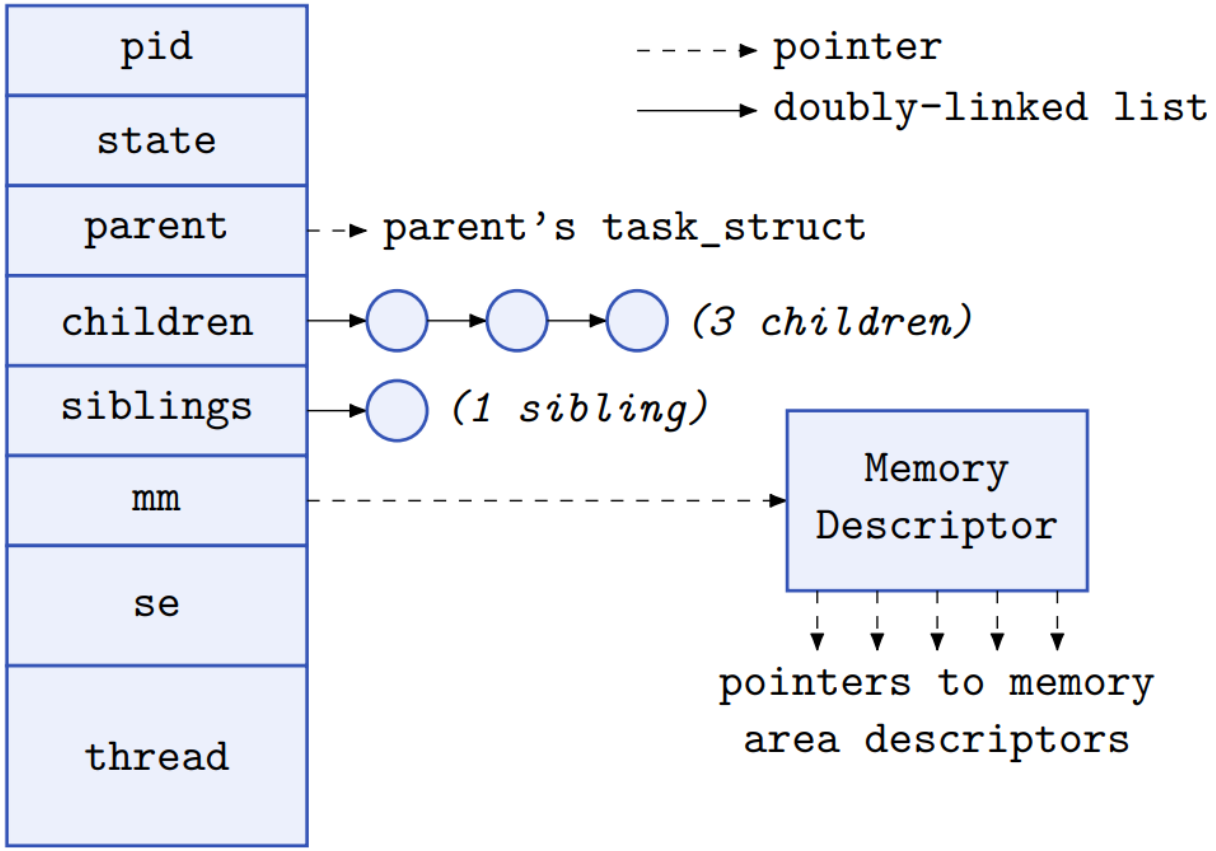
task_struct è una struttura dati usata dal kernel per rappresentare un processo e per salvare le sue informazioni.

La definizione della struct **task_struct** si trova in *mentos/inc/process/process.h* .

```
81  typedef struct task_struct {
83      pid_t pid;
85      pid_t sid;
87      pid_t pgid;
89      pid_t gid;
91      pid_t uid;
92      // -1 unrunnable, 0 runnable, >0 stopped.
94      __volatile__ long state;
96      vfs_file_descriptor_t *fd_list;
98      int max_fd;
100     struct task_struct *parent;
102     list_head run_list;
104     list_head children;
106     list_head sibling;
108     thread_struct_t thread;
110     sched_entity_t se;
112     int exit_code;
114     char name[TASK_NAME_MAX_LENGTH];
116     mm_struct_t *mm;
118     int error_no;
120     char cwd[PATH_MAX];
121
123     uint32_t sigreturn_addr;
125     sighand_t sighand;
127     sigset_t blocked;
129     sigset_t real_blocked;
131     sigset_t saved_sigmask;
133     sigpending_t pending;
134
136     struct timer_list *real_timer;
137
139     unsigned long it_real_incr;
141     unsigned long it_real_value;
143     unsigned long it_virt_incr;
145     unsigned long it_virt_value;
147     unsigned long it_prof_incr;
149     unsigned long it_prof_value;
150
152     termios_t termios;
154     fs_rb_scancode_t keyboard_rb;
155
156     //==== Future work ===========================================================
157     // - task's attributes:
158     // struct task_struct __rcu *real_parent;
159     // int exit_state;
160     // int exit_signal;
161     // struct thread_info thread_info;
162     //============================================================================
163 } task_struct;
```

PROCESS DESCRIPTOR

Questa immagine qua sotto serve a farci vedere come un processo viene rappresentato nella memoria in MentOS.



PROCESS IDENTIFIER

Ora andremo a descrivere cosa rappresentano i parametri più importanti che si trovano all'interno di **task_struct**.

Il **Process identifier** è un ID univoco che identifica un nuovo progetto. All'atto della creazione viene sommato 1 all'ultimo PID assegnato.

In Linux il massimo valore per PID è 32768.

La macro **RESERVED_PID** è definita per riservare un numero di PID per i processi di sistema e daemons, quindi di conseguenza tutti i processi utente hanno un PID più grande del **RESERVED_PID**.

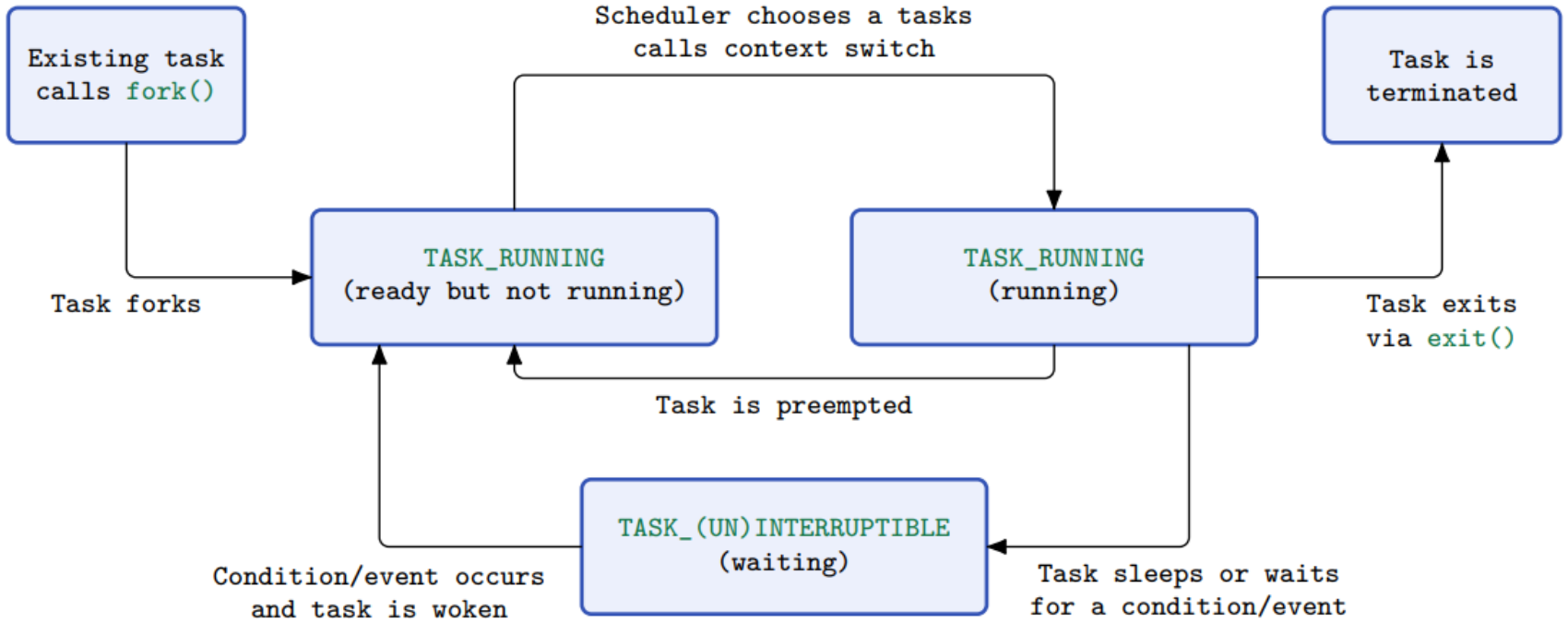
PROCESS STATE

Il **Process State** descrive lo stato corrente di un processo attraverso un valore numerico. Uno stato può quindi trovarsi in uno di questi stati:

- **TASK_RUNNING**: Il processo è attualmente in esecuzione o ha tutte le risorse pronte all'esecuzione a parte la CPU
- **TASK_INTERRUPTIBLE**: il processo è messo in sleep aspettando qualche condizione per essere eseguita. Quando questa condizione esiste, lo stato viene cambiato a TASK_RUNNABLE . Quando incontriamo poi la condizione lo stato si risveglia.
- **TASK_UNINTERRUPTIBLE**: è identica allo stato precedente, solo che in questo caso non dipende da un segnale, bensì il processo deve aspettare senza interruzioni per una specifica wake-up call.
- **TASK_STOPPED**: l'esecuzione del processo è fermata, quindi la task non è in esecuzione o non ha i requisiti per esserelo.
- **EXIT_ZOMBIE**: l'esecuzione è terminata,, ma il parent process non ha fornito una system call wait4(0) o waitpid() per fare la return delle informazioni riguardo al processo morto.
- **EXIT_DIED**: È lo stato finale. Il parent process ha fatto fornito una system call tra wait4(0) o waitpid().

PROCESS STATE

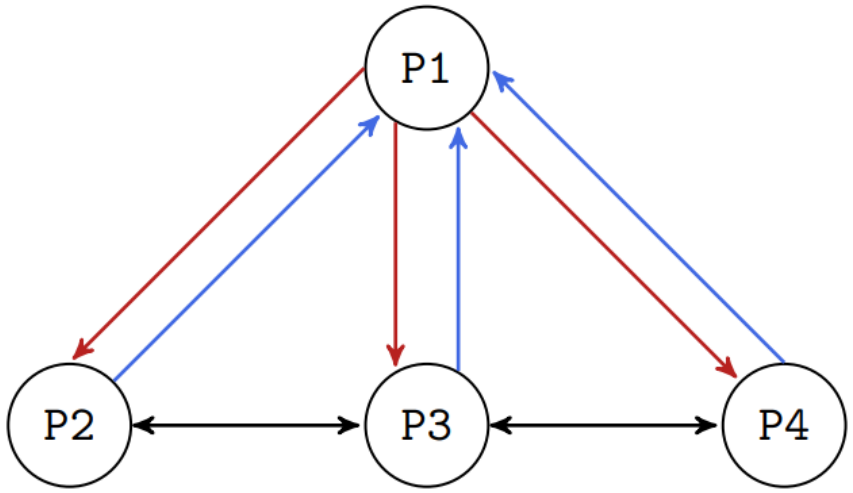
Questo è un flow chart rappresentante tutti i vari process state e come si può passare da uno all'altro.



RELAZIONI TRA I PROCESSI

I processi possono avere una relationship padre/figlio. Quando un processo genera molteplici figli questi hanno una relationship tra fratelli. I campi di **task_struct** qui di seguito servono per:

- **struct task_struct *parent**: puntatore al processo padre
- **struct list_head children**: l'head della lista contenente tutti i children creati dal processo
- **struct list_head siblings**: l'head della lista contenente tutti i children creati dal processo padre.



In **Rosso** abbiamo le relationship da padre a figlio.
In **Blu** abbiamo le relatioship da figlio a padre.
In **Nero** abbiamo le relatioship tra figlio e figlio.

TIME ACCOUNTING

Il campo **se** di task_struct è una struttura **sched_entity** che tiene tutte le informazioni riguardo le attività di scheduling.

La definizione di questa struct si trova a sua volta in *mentos/inc/process/process.h* .

```
31 typedef struct sched_entity_t {
33     int prio;
34
36     time_t start_runtime;
38     time_t exec_start;
40     time_t exec_runtime;
42     time_t sum_exec_runtime;
44     time_t vruntime;
45
47     time_t period;
49     time_t deadline;
51     time_t arrivaltime;
53     bool_t executed;
55     bool_t is_periodic;
57     bool_t is_under_analysis;
59     time_t next_period;
61     time_t worst_case_exec;
63     double utilization_factor;
64 } sched_entity_t;
```

TIME ACCOUNTING PARAMETERS

Analizziamo ora i parametri più importanti di **sched_entity**:

- **prio**: definisce la priorità di un processo. Con un valore che va da [100,139], più basso il numero più è alta la priorità. Di default la prio di un nuovo processo è 120, ma tramite la system call nice(int), dove int è un numero che va da [-20,19] che andrà ad incrementare o decrementare quindi il valore di prio.
- **start_runtime**: il tempo di esecuzione del sistema quando il processo è stato eseguito per la prima volta nella CPU
- **exec_start**: il tempo di esecuzione del sistema quando il processo è stato eseguito per l'ultima volta dalla CPU
- **sum_exec_runtine**: il tempo totale di esecuzione speso dal processo nella CPU
- **vruntime**: la virtual runtime, ovvero la somma pesata del tempo di esecuzione del processo nella CPU.

Inoltre ci sono altri parametri utilizzati come statistiche dei processi.

CONTESTO DI UN PROCESSO

Il campo **thread** di **task_struct** è a sua volta una struttura che contiene le informazioni riguardo l'esecuzione di un processo.

Si tratta del contesto di un processo, e ogni qual volta il processo non è in esecuzione, contiene tutte le informazioni vitali per poter riprenderla.

Contiene infatti tutte le informazioni riguardo puntatori ai registri e allo stack.

SCHEDULER DATA STRUCTURES

La data structure più importante dello scheduler è **runqueue**, in quanto collezione tutti i processi del sistema in esecuzione. La struttura viene definita in *mentos/inc/process/process.h* .

```
13 typedef struct runqueue_t {
15     size_t num_active;
17     size_t num_periodic;
19     list_head queue;
21     task_struct *curr;
22 } runqueue_t;
```

SCHEDULER DATA STRUCTURES

- **num_active**: una size_t , ovvero un long, con il numero di processi che si trovano nel running state.
- **num_priodic**: una size_t , ovvero un long, con il numero di processi periodici che si trovano nel running state.
- ***curr**: una task_struct la quale è un puntatore al processo attualmente in esecuzione.
- **queue**: una list_head_t, una struct che è una lista di tutti i processi in esecuzione.

Il campo queue è la testa di una lista circolare doubly-linked.

EXECUTION FLOW

Le seguenti operazioni sono eseguite dallo scheduler, il quel è chiamato dopo l'handle di interrupt/exception:

1. Aggiorna le variabili temporali del processo corrente;
2. Prova a risvegliare un processo in attesa. Se le condizioni sono rispettate, il processo viene risvegliato e il suo stato viene cambiato a running, inserendolo nella runqueue;
3. Viene eseguito l'algoritmo di scheduling per scegliere il prossimo processo da eseguire dalla runqueue;
4. Viene eseguito un context switch.

ALGORITMI DI SCHEDULING

- Dopo questa breve premessa sulla gestione dei processi, possiamo finalmente parlare di come gli algoritmi di scheduling vengono implementati in MentOS.
- Intanto MentOS supporta vari tipi di algoritmi di scheduling, i quali sono selezionati durante la compilazione tramite cmake e chiamati dalla funzione **scheduler_pick_next_task**;
- Visto che le task all'interno di **runqueue** possono sia essere periodiche che aperiodiche, avremo anche algoritmi per il Real-Time scheduling, ma qui noi ci concentreremo ad analizzare gli algoritmi aperiodici.

ALGORITMI DI SCHEDULING

La funzione centralizzata **scheduler_pick_next_task** è utilizzata dallo scheduler per ottenere il prossimo processo da eseguire. Questa funzione chiamerà internamente l'algoritmo di scheduling che è stato selezionato e in base all' algoritmo verrà scelta la prossimo processo da eseguire.

MentOS supporta per ora tre algoritmi aperiodici:

- **Round Robin** : chiamato con la funzione **__scheduler_rr**;
- **Highest Priority First**: chiamato con la funzione **__scheduler_priority**;
- **Complitely Fair Scheduler**: chiamato con la funzione **__scheduler_cfs**.

La funzione si trova in *mentos/src/process/scheduler_algorithm.c,* dove a loro volta vengono anche definiti tutti i vari algoritmi di scheduling che ci sono.

ALGORITMI DI SCHEDULING

```
185 task_struct *scheduler_pick_next_task(runqueue_t *runqueue)
186 {
187     // Update task statistics.
188     __update_task_statistics(runqueue->curr);
189
190     // Pointer to the next task to schedule.
191     task_struct *next = NULL;
192 #if defined(SCHEDULER_RR)
193     next = __scheduler_rr(runqueue, false);
194 #elif defined(SCHEDULER_PRIORITY)
195     next = __scheduler_priority(runqueue, false);
196 #elif defined(SCHEDULER_CFS)
197     next = __scheduler_cfs(runqueue, false);
198 #elif defined(SCHEDULER_EDF)
199     next = __scheduler_edf(runqueue);
200 #elif defined(SCHEDULER_RM)
201     next = __scheduler_rm(runqueue);
202 #elif defined(SCHEDULER_AEDF)
203     next = __scheduler_aedf(runqueue);
204 #else
205 #error "You should enable a scheduling algorithm!"
206 #endif
207
208     assert(next && "No valid task selected by the scheduling algorithm.");
209
210     // Update the last context switch time of the next task.
211     next->se.exec_start = timer_get_ticks();
212
213     return next;
214 }
```

Se l'algoritmo non viene definito il programma in questione non eseguirà i processi in quanto alla fine degli else if restituiremo un errore. Non c'è implementato quindi un algoritmo di default. Infatti **scheduler_pick_nest_task** sceglierà l'algoritmo in base all'opzione cmake selezionata.

ROUND ROBIN

Il **Round Robin** è un algoritmo di scheduling dove ad ogni processo viene assegnato un slice di tempo fisso in modo ciclico. I suoi pregi sono che è l'algoritmo più facile da implementare, preventivo e soprattutto starvation-free , ovvero si evita che ci siano processi che non vengono mai eseguiti.

Alla funzione si passeranno come parametri il processo corrente e la lista di processi. Si controlla se ci sono altri processi nella lista: se ci sono si va a controllare il successivo processo, il quale deve prima essere eseguibile e poi si verifica se è una task periodica. Se la task è periodica si passa alla task successiva nella coda. Infine, una volta raggiunta una task che rispetta tutti i parametri si fa la return di quest'ultima. Se non vengono invece rispettati i parametri si fa una return NULL, mentre se nella lista c'è un solo elemento si fa la return della task corrente.

ROUND ROBIN

```
41 static inline task_struct *__scheduler_rr(runqueue_t *runqueue, bool_t skip_periodic)
42 {
43     // If there is just one task, return it; no need to do anything.
44     if (list_head_size(&runqueue->curr->run_list) <= 1) {
45         return runqueue->curr;
46     }
47     // Search for the next task (we do not start from the head, so INSIDE, skip the head).
48     list_for_each_decl(it, &runqueue->curr->run_list)
49     {
50         // Check if we reached the head of list_head, and skip it.
51         if (it == &runqueue->queue)
52             continue;
53         // Get the current entry.
54         task_struct *entry = list_entry(it, task_struct, run_list);
55         // We consider only runnable processes
56         if (entry->state != TASK_RUNNING)
57             continue;
58         // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
59         if (__is_periodic_task(entry) && skip_periodic)
60             continue;
61         // We have our next entry.
62         return entry;
63     }
64     return NULL;
65 }
```

ROUND ROBIN

Nonostante noi avessimo detto all'inizio che questi algoritmi sono aperiodici, comunque dobbiamo poter considerare la presenza di task periodiche all'interno della coda di esecuzione.

L'unica cosa che il codice farà sarà un ciclo for dove controlleremo se la task è periodica o meno, e se così sarà faremo uno skip e passeremo a controllare ed eseguire la task successiva.

Questa parte nonostante non sia presente viene data come esercizio per i programmatori da dover completare.

HIGH PRIORITY FIRST

L'algoritmo visto in precedenza è il più semplice, in quanto tutte le task sono uguali e a turno verranno eseguite dalla CPU. Ciò però non è quello che vogliamo che accada sempre, in quanto spesso ci capiterà di voler far eseguire prima delle task rispetto alle altre:, come per esempio far eseguire prima le task degli amministratori rispetto a quelli di uno user normale. Per questo motivo si sono introdotti i priority scheduling algorithm, ovvero degli algoritmi dove ciascuna task avrà un livello di priorità che indicherà la sua posizione nella coda di esecuzione.

HIGH PRIORITY FIRST

Ogni processo ha una priorità data da un numero statico, la quale è più alta più è basso il numero

Lo scheduler semplicemente sceglie il processo da eseguire con la priorità più bassa. Non appena nella runqueue appare un processo con una priorità più alta, questo viene sostituito al processo attualmente in esecuzione.

Il vantaggio quindi di questo tipo di algoritmo è quello di poter avere un'esecuzione gerarchica dove l'importanza di ciascun processo viene definita. Il problema che però potremmo riscontrare, e che prima con il Round Robin non avevamo, è quello della starvation, ovvero processi che restano nella lista in attesa di essere eseguiti per tempi troppo lunghi a causa di processi con priorità più alta.

In questo caso, l'intero algoritmo è lasciato come esercizio allo studente, fornendo a quest'ultimo solo il pseudo codice nelle slide della documentazione e lasciando uno scheletro di questo sempre in *mentos/src/process/scheduler_algorithm.c* .

HIGH PRIORITY FIRST

L'algoritmo in questo caso semplicemente, come quello precedente, prende come parametri in ingresso il processo corrente e la lista dei processi per poi fare un ciclo for su quest'ultima. Il ciclo for permette di prendere dalla lista il processo con la priorità più alta, il quale sarà il prossimo ad essere eseguito.

In caso non avessimo definito la SCHDULER_PRIORITY verrà chiamato come algoritmo il Round Robin.

HIGH PRIORITY FIRST

Pseudo codice fornito nella documentazione.

Pseudocode of Highest Priority First.
Require: Current process c, List of processes L
Ensure: Next process n

```
1:  n = c
2:  for all listNode ∈ L do
3:      if !IsTheHead(L, listNode) then
4:          t = list_entry(listNode)
5:          if priority(t) < priority(n) then
6:              n = t
7:          end if
8:      end if
9:  end for
10: return n
```

HIGH PRIORITY FIRST

Definizione dell'algoritmo in *mentos/src/process/scheduler_algorithm.c*

```
76  static inline task_struct *__scheduler_priority(runqueue_t *runqueue, bool_t skip_periodic)
77  {
78  #ifdef SCHEDULER_PRIORITY
79      // Get the first element of the list.
80      task_struct *next = list_entry(runqueue->curr, task_struct, run_list);
81
82      // Get its static priority.
83      time_t min = /*...*/;
84
85      // Search for the task with the smallest static priority.
86      list_for_each_decl(it, &runqueue->curr->run_list)
87      {
88          // Check if we reached the head of list_head, and skip it.
89          if (it == &runqueue->queue)
90              continue;
91          // Get the current entry.
92          task_struct *entry = list_entry(it, task_struct, run_list);
93          // We consider only runnable processes
94          if (entry->state != TASK_RUNNING)
95              continue;
96          // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
97          if (__is_periodic_task(entry) && skip_periodic)
98              continue;
99          // Check if the entry has a lower priority.
100         if (/*...*/) {
101             // Chose the `entry` as the `next` task.
102             /*...*/
103         }
104     }
105     return next;
106 #else
107     return __scheduler_rr(runqueue, skip_periodic);
108 #endif
109 }
```

COMPLETELY FAIR SCHEDULER

Il **Completely Fair Scheduler (CFS)** ha come obiettivo principale quello di prevenire la starvation dei processi. Ciò viene fatto assegnando la CPU in modo equo fra i vari processi. In questo caso non avremo più una priorità, bensì un peso, il quale identifica il tempo totale di esecuzione del processo.

Per capire la porzione di tempo della CPU da dare ad una determinata task, lo scheduler deve conoscere il peso delle task e , soprattutto, quanto vale tale numero in tempo. Per questo il priority number è mappato ad un determinato peso. Qui di seguito mettiamo la tabella dell'array **prio_to_weight**.

COMPLETELY FAIR SCHEDULER

La tabella si trova in *mentos/inc/process/prio.h* .

```
56 static const int prio_to_weight[NICE_WIDTH] = {
57     /* 100 */ 88761, 71755, 56483, 46273, 36291,
58     /* 105 */ 29154, 23254, 18705, 14949, 11916,
59     /* 110 */ 9548, 7620, 6100, 4904, 3906,
60     /* 115 */ 3121, 2501, 1991, 1586, 1277,
61     /* 120 */ 1024, 820, 655, 526, 423,
62     /* 125 */ 335, 272, 215, 172, 137,
63     /* 130 */ 110, 87, 70, 56, 45,
64     /* 135 */ 36, 29, 23, 18, 15
65 };
```

COMPLETELY FAIR SCHEDULER

Il funzionamento dell'algoritmo è identico a quello del HPF con l'unica differenza che al posto della priorità abbiamo un parametro dato dalla seguente formula:

vruntime +* delta_exec * (NICE_0_LOAD / weight(p))

Dove:

- **vruntime** è il tempo virtuale del processo;
- **delta_exec** è l'ultimo tempo speso dal processo p nella CPU;
- **NICE_0_LOAD** è il peso della task con priorità 120 (il quale numero equivale a 1024);
- **weight(p)** è il peso di p identificato dall'array **prio_to_weight**.

Come il caso precedente anche di questa funzione abbiamo lo scheletro e ci viene fornito nelle slide uno pseudo codice che ci aiuterà nell' implmentazione.

COMPLETELY FAIR SCHEDULER

Definizione dell'algoritmo in *mentos/src/process/scheduler_algorithm.c*

```
120 static inline task_struct *__scheduler_cfs(runqueue_t *runqueue, bool_t skip_periodic)
121 {
122 #ifdef SCHEDULER_CFS
123     // Get the first element of the list.
124     task_struct *next = list_entry(runqueue->curr, task_struct, run_list);
125 
126     // Get its virtual runtime.
127     time_t min = /* ... */;
128 
129     // Search for the task with the smallest vruntime value.
130     list_for_each_decl(it, &runqueue->curr->run_list)
131     {
132         // Check if we reached the head of list_head, and skip it.
133         if (it == &runqueue->queue)
134             continue;
135         // Get the current entry.
136         task_struct *entry = list_entry(it, task_struct, run_list);
137         // We consider only runnable processes
138         if (entry->state != TASK_RUNNING)
139             continue;
140         // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
141         if (__is_periodic_task(entry) && skip_periodic)
142             continue;
143 
144         // Check if the element in the list has a smaller vruntime value.
145         /* ... */
146     }
147     return next;
148 #else
149     return __scheduler_rr(runqueue, skip_periodic);
150 #endif
151 }
```

COMPLETELY FAIR SCHEDULER

Pseudo codice fornito nella documentazione.

Pseudocode of Completely Fair Scheduler.

Require: Current process c, List of processes L

Ensure: Next process n

```
1:  updateVirtualRuntime(c)
2:  n = c
3:  for all listNode ∈ L do
4:      if !IsTheHead(L, listNode) then
5:          task = list_entry(listNode)
6:          if virtualRuntime(task) < virtualRuntime(n) then
7:              n = task
8:          end if
9:      end if
10: end for
11: return n
```

GESTIONE DEI PROCESSI IN OS161

La gestione dei processi in OS161 è molto primitiva e molte parti vengono lasciate da completare. Rispetto a MentOS abbiamo molti parametri in meno e la gestione è molto semplificata lasciando all'interno della classe **proc** solo i dati essenziali per la gestione di quest'ultimi.

Gli elementi della classe saranno:

- ***p_name**: il nome del processo
- **p_lock**: il lock di questa struttura
- **p_numthreads**: il numero di thread di questo processo
- ***p_addrspace**: il virtual address space
- ***p_cwd**: la working directory attuale del processo

GESTIONE DEI PROCESSI IN OS161

Ecco la **struct proc** che si trova in *kern/include/proc.h* .

```
62  struct proc {
63      char *p_name;            /* Name of this process */
64      struct spinlock p_lock;     /* Lock for this structure */
65      unsigned p_numthreads;      /* Number of threads in this process */
66
67      /* VM */
68      struct addrspace *p_addrspace;  /* virtual address space */
69
70      /* VFS */
71      struct vnode *p_cwd;        /* current working directory */
72
73      /* add more material here as needed */
74  };
```

ALGORITMI DI SCHEDULING IN OS161

Altrettanto semplice risulta la gestione degli algoritmi di scheduling in OS161. in quanto l'unico algoritmo che risulta essere implementato è un semplice Round Robin il quale viene gestito tramite la funzione **hardclock** che si trova in *kern/thread/clock.c*. L'algoritmo che si occuperà dell'ordinamento delle task è **schedule** il quale è definito all'interno di *kern/thread/thread.c* . La funzione **schedule** in questo caso non farà nulla, quindi la coda di esecuzione rimarrà invariata seguendo quindi un'esecuzione in ordine di ingresso, quindi funzionando come un Round Robin.

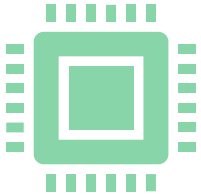
Notiamo quindi che l'implementazione seguente non permette di poter switchare tra vari possibili algoritmi ma ci permette solamente di poterne utilizzare uno solo.

ALGORITMI DI SCHEDULING IN OS161

```
92  void
93  hardclock(void)
94  {
95      /*
96       * Collect statistics here as desired.
97       */
98
99      curcpu->c_hardclocks++;
100     if ((curcpu->c_hardclocks % MIGRATE_HARDCLOCKS) == 0) {
101         thread_consider_migration();
102     }
103     if ((curcpu->c_hardclocks % SCHEDULE_HARDCLOCKS) == 0) {
104         schedule();
105     }
106     thread_yield();
107 }
```

```
821 void
822 schedule(void)
823 {
824     /*
825      * You can write this. If we do nothing, threads will run in
826      * round-robin fashion.
827      */
828 }
```

SOURCES & CREDITS



Sources:

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

OS/161: http://www.os161.org/

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