SIMULATORE DI CHIAMATE A PROCEDURA.

Esercizio del progetto numero 1

Data consegna: 03 luglio 2016

Autori:

Salani Lorenzo: lorenzo.salani@stud.unifi.it

Fagioli Giulio: giulio.fagioli@stud.unifi.it

Buonanno Cecilia: cecilia.buonanno@stud.unifi.it

Sommario

Descrizione dell'esercizio.

Il testo dell'esercizio richiedeva di scrivere un programma in assembly MIPS che, data una stringa S che rappresentasse una qualsiasi combinazione delle funzioni somma, sottrazione, prodotto e divisione intera fra due numeri interi, prenda in input la stringa S letta da un file di massimo 150 caratteri (da noi chiamato "chiamate.txt"), e che:
1. implementi le funzioni somma, sottrazione, prodotto e divisione intera fra due numeri interi, come procedure MIPS;
2. simuli la sequenza delle chiamate alle funzioni da sinistra a destra, come appaiono nella stringa in input;
3.visualizzi su console la traccia con la sequenza delle chiamate annidate (con argomento fra parentesi) ed i valori restituiti dalle varie chiamate annidate (fra parentesi).

Inoltre, era necessario mostrare e discutere nella relazione l'evoluzione dello stack nel caso specifico in cui la stringa sarebbe stata S= "somma(somma(1,2),somma(3,4))", questo, nella relazione stessa.

Un esempio di output su console, nel caso in cui S= "divisione(5,2)":
--> divisione(5,2)
<-- divisione-return(2)
(Infatti la divisione5/2=2) poiché c'è da ricordarsi che si tratta di divisione intera, infatti i numeri che verrebbero dopo la virgola non vengono considerati.

Soluzione adottata.

In questo capitolo faremo riferimento unicamente alla modalità utilizzata per la risoluzione di tale esercizio in linguaggio naturale ed inserendo immagini per una migliore comprensione.

La nostra idea è stata quella di inserire, in principio, le operazioni in 4 indirizzi differenti del registro $t0 e poi far leggere al programma la stringa immessa nel file “chiamate.txt”, in modo che il programma la acquisisca per lavorarci, e salvarla in un registro, in questo caso in $t6 e poi caricare il descrittore del file in $a0.

Abbiamo inoltre inizializzato una variabile indice globale (idx) che ci aiuta a scorrere la stringa ed, in base alla lettera trovata, ci aiuti a definire il tipo di operazione da svolgere.
Il contatore idx parte da 2, poiché la prima lettera utile per verificare il tipo di operazione è la terza, in questo caso appunto abbiamo usato uno “switch”, nel caso in cui la lettera trovata fosse una “m” allora si ha una somma, se fosse una ”o” allora sarebbe un prodotto, se fosse una “t” allora si esegue una sottrazione, altrimenti se fosse “v” una divisione.

Quando l’indice trova una lettera allora cerca la prima parentesi aperta “(“ sul suo percorso e grazie ad un’altra procedura trova la fine degli argomenti all’interno della parentesi già aperta, cercando infatti il carattere della parentesi chiusa “)”.
Se appunto all’interno trova un’altra lettera, esegue le stesse operazioni sopraelencate dopo aver salvato la fine di quest’altra stringa ed averla stampata, mentre se trova un numero, dopo aver salvato questo ed il successivo in due registri diversi, stampa l’operazione del return richiesta fatta tra i due numeri trovati tra parentesi.

Questa operazione viene fatta tante volte quante sono le operazioni da fare
Sono presenti molte funzioni che hanno compiti molto diversi e più complicati, i quali spiegheremo più approfonditamente nel prossimo capitolo, analizzando perfino il codice con i rispettivi commenti.

Analisi del codice.

In questo capitolo verrà analizzato tutto il codice in assembly, le varie procedure, i registri, le variabili utilizzate e tutte le operazioni in modo dettagliato.

```
.data

jump_table: .space 16

p1: .asciiz  "Trovata una lerrera"
cont:   .asciiz  "File contents: "
errore: .asciiz  "File contents: "

buffer:    .space 1024
idx:       .word 2
indent:    .word 0
```

Il seguente codice è l'inizio del nostro progetto.

La *jump_table* viene inizializzata come un array di 4 word che verrà instanziato dal main con gli indirizzi delle label che richiameranno le procedure corrispondenti.

Il campo *buffer* è lo spazio di memoria che conterrà la stringa letta, *idx* invece viene da noi inizializzata come una variabile di indice globale che ci servirà per scorrere la stringa e viene inizializzato a 2, mentre *indent* conterrà poi i numeri di indentazioni (tab) da stampare, infatti è inizializzato a 0.

```
17  #Stringhe di errore
18  divzero: .asciiz "Impossibile eseguire una divisone per zero."
19  fnf:     .ascii  "File non trovato o si è verificato un errore nella lettura di "
20  nomefile: .asciiz "chiamate.txt"        # Nome del file
21
22  #Stringhe di output
23  stampafreccie: .asciiz "--> "
24  stampasommareturn: .asciiz "<-- somma-return("
25  stampaprodottoreturn: .asciiz "<-- prodotto-return("
26  stampasottrazionereturn: .asciiz "<-- sottrazione-return("
27  stampadivisionereturn: .asciiz "<-- divisione-return("
28  parentesi: .asciiz ")"
29  tab : .ascii "  "
30
```

Il frammento di codice dalla riga 17 alla riga 30 identifica tutte le stringhe che verranno stampate se richiamate nel codice con la syscall.

```
31  .text
32  .globl main
33  main:
34          la $t1, jump_table
35          la $t0, somma
36          sw $t0, 0($t1)
37          la $t0, prodotto
38          sw $t0, 4($t1)
39          la $t0, sottrazione
40          sw $t0, 8($t1)
41          la $t0, divisione
42          sw $t0,12($t1)
43  # Apertura del file
44          li      $v0, 13
45          la      $a0, nomefile
46          li      $a1, 0
47          li      $a2, 0
48          syscall
49          move    $t6, $v0
50          blt     $v0, 0, errfile
```

A questo punto abbiamo la *procedura main* in cui viene caricato nel registro $t1 la *jump_table*, ovvero l'area di memoria in cui vengono memorizzati gli indirizzi delle etichette a cui saltare.

A questo punto con l'istruzione *load address* (la) il programma carica gli indirizzi delle procedure nel registro $t0 ed in seguito carica le 4 procedure agli indirizzi corrispettivi della jump_table.

Nel registro $v0 viene caricato 13, che è la funzione che apre un file tramite una syscall, la funzione di apertura del file richiede che sia presente nel registro $a0 il nome del file, nel registro $a1 i flags e in $a2 la modalità per aprire il file. Il registro dei flag è impostato a 0 poiché è un file di lettura mentre quello del mode è impostato a 0 poiché non è necessario specificare la modalità di apertura del file.

In seguito l'indirizzo del file viene spostato in $t6 e l'istruzione seguente controlla che il numero dell'indirizzo sia maggiore di 0, in caso contrario allora il programma passa alla procedura con etichetta "*errfile*".

```
595 # Errori
596 errfile:
597         li      $v0, 4
598         la      $a0, fnf
599         syscall
600         j fine
601
```

In questo caso si fa una chiamata di sistema alla funzione con codice 4, ovvero stampa stringa, che stampa la stringa con etichetta "fnf", tra le stringhe di errore.
Dopo aver fatto questo il programma salta alla fine.

```
51  # Lettura del file
52          li      $v0, 14
53          move    $a0, $t6
54          la      $a1, buffer
55          li      $a2, 1024
56          syscall
57  # Chiusura del file
58          li      $v0, 16
59          move    $a0, $t6
60          syscall
```

Nel caso in cui l'indirizzo sia un numero maggiore di 0, si prosegue leggendo il file, in $a0 viene messo l'argomento presente precedentemente in $t6, che conteneva appunto l'indirizzo del file, in $a1 viene messo l'indirizzo di input del *buffer,* poiché è l'indirizzo di partenza in cui si inserisce la stinga ed in $a2 viene caricato il numero massimo di caratteri da leggere; come risultato in $v0 viene inserito il numero di caratteri letti.

Successivamente è necessario chiudere il file, poiché questo non viene modificato. Viene dunque fatta una chiamata di sistema con codice 16, ovvero quello di chiusura file ed è necessario mettere nel registro $a0 l'indirizzo in cui è presente il file.

Adesso è necessario fare il controllo della prima operazione da eseguire e fare il richiamo della relativa procedura:

```
63          lw $t0, idx
64          li $t1,'a'
65          li $t2,'z'
66
67          la $a1,buffer
68          add $a1,$a1,$t0
69          lb $a0,($a1)
70
71          jal switchoperazione
```

Viene messo l'indice idx inizializzato a 2 nel registro $t0, con l'istruzione *load immediate (li)* inseriamo il carattere 'a' ed il carattere 'z' rispettivamente nei registri $t1 e $t2.
In $a1 viene caricato l'indirizzo del buffer a cui poi con l'istruzione *add* viene sommato l'indice idx ed il risultato viene messo in $a1, in seguito carica il primo byte di $a1 in $a0. Dopo con l'istruzione *jal* il programma salta alla procedura "*switchoperazione*", caricando nel registro $ra il registro della procedura e così saltando all'indirizzo del target scelto.

```
495 switchoperazione:
496         beq    $a0, 'm', jsomma
497         beq    $a0, 'v', jdivisione
498         beq    $a0, 'o', jprodotto
499         beq    $a0, 't', jsottrazione
```

In questa procedura il programma controlla quale tra i seguenti caratteri è presente in $a0 (tramite l'istruzione *branch if equal*), a seconda della lettera il programma salta alla procedura che svolge l'operazione richiesta.

```
500 jsomma:
501         li $t0,0
502         j branch_case
503 jprodotto:
504         li $t0,1
505         j branch_case
506 jsottrazione:
507         li $t0,2
508         j branch_case
509 jdivisione:
510         li $t0,3
511         j branch_case
```

In caso in cui fosse una somma, nel registro $t0 viene caricato il numero 1, che è come se fosse un codice identificativo per poi eseguire l'operazione di somma, e poi salta alla procedura *branch_case,* analogamente avviene con le altre operazioni che però hanno codice 2, 3 o 4 in caso in cui si tratti di un prodotto, di una sottrazione o di una divisione.

```
512 branch_case:
513         add $t0, $t0, $t0
514         add $t0, $t0, $t0
515         la $t1, jump_table
516         add $t0, $t0, $t1
517         lw $t0, 0($t0)
518
519         addi $sp,$sp,-4
520         sw $ra,0($sp)
521
522         jalr $t0
523
524         lw $ra,0($sp)
525         addi $sp,$sp,4
526
527         jr $ra
528
```

Nella *procedura branch_case* si moltiplica il numero trovato precedentemente per 4 e viene messo sempre in $t0, per fare questa operazione abbiamo sommato 4 volte lo stesso registro, facendo due istruzioni *add*, in questo caso sapremo il valore dell'indirizzo cui dobbiamo saltare.
In $t1 viene caricato invece l'indirizzo della *jump_table* e viene sommato al registro $t0.
A questo punto carico in $t0 l'indirizzo trovato al quale devo saltare.

Usando *$sp* (stack pointer) allochiamo una word (4byte) nello stack per memorizzare $ra, questa procedura è necessaria poiché nel programma vengono fatte più chiamate annidate, che lo cancellerebbero. Successivamente con l'istruzione *jalr* saltiamo all'indirizzo contenuto in $t0, dopo questa chiamata viene ricaricato il registro $ra precedentemente salvato nello stack e con l'ultima istruzione si ritorna alla procedura chiamante.
In questo caso il programma tornerà da dove era stata richiamata la funzione nelle prime righe del codice ed avremmo già un risultato che è stato frutto delle procedure che prevedono le varie operazioni, che spiegheremo in seguito.

```
73          move $t0,$v0
74
75          li $a0,10
76          li $v0, 11
77          syscall
78
79          move $a0, $t0
80          li $v0, 1
81          syscall
82
83          j fine
```

A questo punto il contenuto del registro $v0, in cui c'è il risultato dell'operazione, viene spostato nel registro $t0.
In $a0 viene messo il carattere ASCII a capo e in $v0 la funzione 11 delle chiamate di sistema, ovvero stampa carattere, in questo caso così la chiamata di sistema stamperà il continuo del risultato a capo; il risultato che si trovava nell'indirizzo $t0 viene spostato nel registro $a0, poiché la chiamata di sistema che stampa un intero (codice 1, messa in $v0) prevede che nel registro $a0 si abbia il numero da stampare. Il codice poi salta a *fine*, ovvero il programma termina.

Procedura *"somma"*:

```
somma:
        addi $sp,$sp,-4
        sw $ra,0($sp)

        jal stampaTab

        lw $t0,idx
        addi $sp,$sp,-4
        sw $t0,0($sp)

        li $a0,'('
        jal ricerca

        jal trovafine

        move $a1,$v0

        lw $a0,0($sp)
        addi $sp,$sp,4
        subi $a0,$a0,2

        jal stampaIntervallo
        lw $t0, indent
        addi $t0, $t0, 1
        sw $t0, indent

        li $a0,10
        li $v0, 11
        syscall

        jal parse

        add $t1, $v0, $v1

        lw $t0, indent
        subi $t0, $t0, 1
        sw $t0, indent

        jal stampaTab

        lw $ra,0($sp)
        addi $sp,$sp,4
        li      $v0, 4
        la      $a0, stampasomma
        syscall
        li      $v0, 1
        move    $a0, $t1
        syscall
        li      $v0, 4
        la      $a0, parentesi
        syscall
        li $a0,10
        li $v0, 11
        syscall

        move $v0, $t1
        jr $ra
```

Il codice rappresentato è quello della procedura somma, i casi delle altre operazioni sono, tuttavia, molto simili.

Nello stack pointer viene allocata una memoria di 4 byte e poi viene salvato nello stack, per recuperarlo solo dopo altre operazioni, in modo tale da non essere modificabile.
Si richiama la procedura "*stampaTab*", che serve per stampare le indentazioni. Poi viene caricato l'indice idx nel registro $t0, che conteneva la terza lettera dell'operazione trovata. Vengono di nuovo liberati 4 byte di memoria dallo stack pointer per poter inserire l'indice idx nello stack, poiché altrimenti verrebbe modificato nelle prossime procedure.

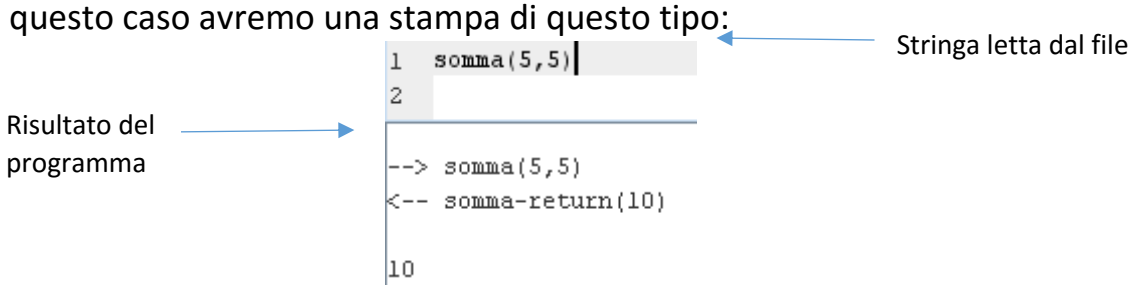
Nel registro $a0 viene inserito il carattere '(' che indica l'inizio dei dati dell'operazione, ed in seguito si effettua una ricerca, sia dell'inizio, che della fine, infatti la procedura *trovafine* restituisce la posizione della parentesi chiusa che trova.
In $a1 viene caricata la posizione finale della stringa.

In $a0 viene ripreso l'indice idx che era stato messo nello stack, poi vengono deallocati 4 byte dallo stack pointer e viene sottratto 2 al registro $a0 che adesso contiene la posizione iniziale della stringa da stampare; il richiamo alla procedura *stampaIntervallo* fa si che venga stampata la stringa data all'inizio.

Dopodiché viene caricata la variabile che conta le indentazioni da fare nel registro $t0 e, dopo aver stampato un intervallo, questa viene aumentata di 1 per poi essere risalvata nella variabile *indent.*

Con una chiamata di sistema viene stampato il carattere per andare a capo e poi viene richiamata la procedura *parse* che spiegheremo nel prossimo capitolo.
Dopo aver eseguito la procedura parse, che legge i valori all'interno delle parentesi e li salva nei registri, viene fatta l'operazione tra i due valori trovati (in questo caso viene fatta una somma) ed il risultato di ritorno viene salvato in $t1.
Le indentazioni nella variabile vengono decrementate di 1 e poi vengono stampate.

Dallo Stack viene recuperato il registro $ra, quello del chiamante e poi vengono deallocati dallo stack pointer 4 byte.
Adesso dobbiamo stampare il valore di ritorno e per farlo si utilizza una chiamata di sistema, infatti viene caricato, con un *load immediate* nel registro $v0 il codice 4, che identifica la funzione stampa stringa e nel registro $a0 viene caricata la stringa da stampare, che avevamo inizializzato all'inizio del programma, in questo caso stampa "*somma-return(".*
Con un'altra chiamata di sistema viene caricato il codice della funzione in $v0, in questo caso 1 perché dovrà stampare un numero intero, e in $a0 viene caricato il registro $t1, che conteneva il risultato dell'operazione, poi viene fatta un'altra chiamata di sistema per stampare la stringa ")", perciò viene caricato nel registro $v0 il codice 4 e nel registro $a0 viene caricata la stringa.
In questo caso avremo una stampa di questo tipo:



Viene nuovamente stampato il carattere per andare a capo e poi ricaricato il risultato dell'operazione che si trovava in $t0, in $v0 e si risalta al registro del chiamante.

Le procedure delle altre operazioni verranno spiegate più superficialmente poiché molte righe del codice sono simili o addirittura uguali a quelle della somma, verranno spiegate solo le parti in più e le differenze.

Procedura *prodotto*:

```
145 prodotto:
146
147         addi $sp,$sp,-4
148         sw $ra,0($sp)
149
150         jal stampaTab
151
152         lw $t0,idx
153         addi $sp,$sp,-4
154         sw $t0,0($sp)
155
156         li $a0,'('
157         jal ricerca
158
159         jal trovafine
160
161         move $a1,$v0
162
163         lw $a0,0($sp)
164         addi $sp,$sp,4
165         subi $a0,$a0,2
166
167         jal stampaIntervallo
```

```
169         lw $t0, indent
170         addi $t0, $t0, 1
171         sw $t0, indent
172
173         li $a0,10
174         li $v0, 11
175         syscall
176
177         jal parse
178
179         mul $t1, $v0, $v1
180
181         lw $t0, indent
182         subi $t0, $t0, 1
183         sw $t0, indent
184
185         jal stampaTab
186
187         lw $ra,0($sp)
188         addi $sp,$sp,4
```

```
189
190         li      $v0, 4
191         la      $a0, stampaprodottoreturr
192         syscall
193         li      $v0, 1
194         move    $a0, $t1
195         syscall
196         li      $v0, 4
197         la      $a0, parentesi
198         syscall
199         li $a0,10
200         li $v0, 11
201         syscall
202
203         move $v0, $t1
204         jr $ra
205
206
```

La procedura del prodotto è simile a quella della somma, cambia la riga 179 in cui al posto dell'operazione add tra i registri dei due numeri trovati, si ha l'operazione *mul* che mette nel registro $t1 il prodotto dei numeri contenuti nei registri $v0 e $v1. L'unica altra differenza è la stringa stampata nel return, in questo caso quando si fa la chiamata di sistema la stringa caricata nel registro $a0 (riga 191), è quella del prodotto, che appunto sarà in questo modo:

```
1  prodotto(2,5)
2
```

Stringa letta dal file

Stampa del programma

```
--> prodotto(2,5)
<-- prodotto-return(10)

10
```

Procedura *sottrazione*:

```
207  sottrazione:
208
209      addi $sp,$sp,-4
210      sw $ra,0($sp)
211
212      jal stampaTab
213
214      lw $t0,idx
215      addi $sp,$sp,-4
216      sw $t0,0($sp)
217
218      li $a0,'('
219      jal ricerca
220
221      jal trovafine
222
223      move $a1,$v0
224
225      lw $a0,0($sp)
226      addi $sp,$sp,4
227      subi $a0,$a0,2
228
229      jal stampaIntervallo
```

```
231      lw $t0, indent
232      addi $t0, $t0, 1
233      sw $t0, indent
234
235      li $a0,10
236      li $v0, 11
237      syscall
238
239      jal parse
240
241      sub $t1, $v0, $v1
242
243      lw $t0, indent
244      subi $t0, $t0, 1
245      sw $t0, indent
246
247      jal stampaTab
248
249      lw $ra,0($sp)
250      addi $sp,$sp,4
```

```
249  lw $ra,0($sp)
250  addi $sp,$sp,4
251
252  li      $v0, 4
253  la      $a0, stampasottrazionereturn
254  syscall
255  li      $v0, 1
256  move    $a0, $t1
257  syscall
258  li      $v0, 4
259  la      $a0, parentesi
260  syscall
261  li $a0,10
262  li $v0, 11
263  syscall
264
265  move $v0, $t1
266  jr $ra
```

Anche nella procedura della sottrazione le differenze sono le stesse, nella riga 241 si ha l'operazione *sub* che effettua la sottrazione tra i registri $v0 e $v1 e viene messo il risultato in $t1 e poi alla riga 253 viene stampata la stringa del return della sottrazione come vedremo nell'esempio seguente.

```
1  sottrazione(6,3)
2
--> sottrazione(6,3)
<-- sottrazione-return(3)

3
```

```
1  sottrazione(3,6)
2
--> sottrazione(3,6)
<-- sottrazione-return(-3)

-3
```

Procedura *divisione*:

```
268  divisione:
269
270      addi $sp,$sp,-4
271      sw $ra,0($sp)
272
273      jal stampaTab
274
275      lw $t0,idx
276      addi $sp,$sp,-4
277      sw $t0,0($sp)
278
279      li $a0,'('
280      jal ricerca
281
282      jal trovafine
283
284      move $a1,$v0
285
286      lw $a0,0($sp)
287      addi $sp,$sp,4
288      subi $a0,$a0,2
289
290      jal stampaIntervallo
```

```
292      lw $t0, indent
293      addi $t0, $t0, 1
294      sw $t0, indent
295
296      li $a0,10
297      li $v0, 11
298      syscall
299
300      jal parse
301
302      beq $v1, 0, errdivzero
303
304
305      div $t1, $v0, $v1
306
307      lw $t0, indent
308      subi $t0, $t0, 1
309      sw $t0, indent
310
311      jal stampaTab
```

```
313      lw $ra,0($sp)
314      addi $sp,$sp,4
315
316      li      $v0, 4
317      la      $a0, stampadivisionereturn
318      syscall
319      li      $v0, 1
320      move    $a0, $t1
321      syscall
322      li      $v0, 4
323      la      $a0, parentesi
324      syscall
325      li $a0,10
326      li $v0, 11
327      syscall
328
329      move $v0, $t1
330      jr $ra
```

Anche la procedura della divisione è molto simile alle altre; prima di svolgere l'operazione *div* tra $v0 e $v1, faremo il controllo sul contenuto del registro $v1, il numero contenuto in esso dovrà essere diverso da 0, poiché sappiamo che non è possibile dividere nessun numero per 0.
Dopo aver letto i due numeri con la procedura del *parse,* con l'operazione *beq* controlleremo se il contenuto del registro $v1 è uguale a 0, se questo è vero allora si passerà alla procedura *errdivzero* che stamperà la stringa di errore ed il programma finirà.

Inoltre come negli altri casi la stampa del return cambia, si avrà infatti quella della divisione.

N.B. la divisione effettuata è la divisione intera!

```
1  divisione(4,2)

--> divisione(4,2)
<-- divisione-return(2)

2
```

```
1  divisione(5,2)

--> divisione(5,2)
<-- divisione-return(2)

2
```

```
1  divisione(5,0)

--> divisione(5,0)
Impossibile eseguire una divisone per zero.
-- program is finished running --
```

Procedura *parse*:

```
331
332  parse:
333       lw $t0,idx
334       la $a1,buffer
335       add $a1,$a1,$t0
336       lb $a0,($a1)
337
338       li $t3,0
339       beq $a0,'+', primoPiu
340       beq $a0,'-', primoMen
341       blt $a0,'0', primalettera
342       bgt $a0,'9', primalettera
343       j primonumero
344       primoMen:
345       li $t3,1
346       primoPiu:
347       addi $a1, $a1, 1
```

La procedura parse ci permette di leggere i due numeri tra cui verrà eseguita un'operazione, quindi viene caricato in $t0 l'indice idx ed in $a1 l'indirizzo di buffer, questi vengono sommati e caricati in $a1 ed il registro $a0 prende il primo byte del registro $a1.

Nel registro $t3 viene caricato il valore 0 per controllare il segno.
Adesso si controlla cosa c'è nel registro $a0, se è presente un '+' il programma passa alla procedura *primoPiu* che carica in $a1 la somma tra $a1 ed 1, se vi è presente un '-' allora passa alla procedura *primoMen* che carica in $t3 il numero 1, se in $a0 c'è un valore più piccolo di 0 o più grande di 9, salta alla procedura *primalettera,* altrimenti se non è nessuno dei casi già trovati, salta alla procedura *primonumero.*

```
349  primonumero:
350          li $t0,10
351          li $s2,0
```

Qui viene caricato il valore 10 in $t0 per essere usato nel ciclo successivo e viene azzerato il registro$s2.

Viene aggiunto il valore 2 al registro $t0 con un'operazione di tipo *add immediate* e poi caricato in idx.

```
373  primalettera:
374
375          addi $t0,$t0,2
376          sw $t0,idx
377
378          addi $sp,$sp,-4
379          sw $ra,0($sp)
380
381          lw $t0,idx
382          la $a1,buffer
383          add $a1,$a1,$t0
384          lb $a0,($a1)
385
386          jal switchoperazione
387
388          lw $ra,0($sp)
389          addi $sp,$sp,4
390
391          addi $sp,$sp,-4
392          sw $v0,0($sp)
```

Poi vengono liberati 4 byte dallo stack pointer per poi esser caricato nello stack in modo da non poter essere modificato.

Viene caricato idx in $t0 ed il buffer letto in $a1, questi vengono sommati e caricati nel registro $a1 e poi il registro $a0 acquisisce il primo byte di $a1.

Si passa poi alla procedura switchoperazione.
Viene recuperato $ra dallo stack e vengono deallocati 4 byte nello stack pointer.

Successivamente vengono allocati 4 byte nello stack pointer e viene salvato il risultato dell'operazione eseguita nella funzione switchoperazione nello stack per evitare che si perda il risultato dell'operazione.

```
352  caricacifre1:
353          lb $t1, ($a1)
354          blt $t1, 48, cifrecaricate1
355          bgt $t1, 57, cifrecaricate1
356          addi $t1, $t1, -48
357          mul $s2, $s2, $t0
358          add $s2, $s2, $t1
359          addi $a1, $a1, 1
360          j caricacifre1
361   cifrecaricate1:
362
363          beq $t3,0, primoPositivo
364          sub $s2,$zero,$s2
```

La procedura *caricacifre1* carica il primo byte della cifra trovata in $t1, se $t1 non è un numero, ovvero compreso tra il codice Ascii 48 e 57, tutte le cifre sono state caricate, quindi si esce e si passa a cifrecaricate1. Viene riconvertito $t1 da Ascii in decimale e si moltiplica il numero per 10.
Viene aggiunta la cifra al numero come unità, viene incrementato l'indirizzo di buffer e poi si salta all'etichetta caricacifre1.La procedura *cifrecaricate1* verifica se il contenuto del registro $t3 è uguale a 0 significa che il numero non è preceduto da un '-' e perciò si prosegue la lettura andando all'etichetta primoPositivo, altrimenti nel registro dove era contenuto il numero, mettiamo la sottrazione tra 0 ed il numero stesso, rendendolo così negativo.

```
366  primoPositivo:
367
368          addi $sp,$sp,-4
369          sw $s2,0($sp)
370
371          j secondocarattere
```

Vengono deallocati 4 bytes dallo stack pointer, in seguito viene caricato il contenuto del registro $s2, in cui c'era il numero trovato, nello stack, poi si salta alla procedura secondocarattere.

Gestione del secondo carattere:

```
394  secondocarattere:
395          addi $sp,$sp,-4
396          sw $ra,0($sp)
397
398          li $a0,','
399          jal ricerca
400
401          lw $ra,0($sp)
402          addi $sp,$sp,4
403
404          lw $t0,idx
405
406          la $a1,buffer
407          add $a1,$a1,$t0
408          lb $a0,($a1)
409
410          li $t3,0
411          beq $a0,'+', secondoPiu
412          beq $a0,'-', secondoMeno
413          blt $a0,'0', secondaLettera
414          bgt $a0,'9', secondaLettera
415          j secondoNumero
```

Vengono deallocati 4 byte dallo stack pointer e poi viene caricato il registro del chiamante nello stack per non essere modificato.
In seguito viene caricato il carattere virgola (',') nel registro $a0 e si salta alla procedura "ricerca".
Dopo aver fatto la ricerca si ricarica in $ra il valore salvato precedentemente e dopo vengono allocati 4 byte nello stack.
Viene poi caricato l'indice idx nel registro $t0 e l'indirizzo del buffer nel registro $a1, a questo viene aggiunto il contenuto del registro $t0 e nel registro $a0 viene caricato il primo byte del registro $a1.
Viene caricato il valore 0 nel registro $t3 e, come nella procedura del controllo del primocarattere, a seconda del carattere presente in $a0 si salta a differenti procedure o in caso sia un numero, si salta alla procedura secondoNumero.

```
416 secondoMeno:
417         li $t3,1
418 secondoPiu:
419         addi $a1, $a1, 1
420
421
422 secondoNumero:
423
424         li $t0,10
425         li $s2,0
```

La procedura *secondoMeno* carica in $t3 il numero 1.

La procedura *secondoPiu* somma il valore 1 al contenuto del registro $a1.

La procedura *secondoNumero* carica il valore 10 in $t0 per essere poi utilizzato nella procedura successiva e viene azzerato il registro $s2.

```
426 caricacifre2:
427         lb $t1, ($a1)
428         blt $t1, 48, cifrecaricate2
429         bgt $t1, 57, cifrecaricate2
430         addi $t1, $t1, -48
431         mul $s2, $s2, $t0
432         add $s2, $s2, $t1
433         addi $a1, $a1, 1
434         j caricacifre2
435  cifrecaricate2:
436
437         beq $t3,0, secondoPositivo
438         sub $s2,$zero,$s2
```

La procedura *caricacifre2* e la procedura *cifrecaricate2* hanno la stessa struttura di "caricacifre1" e "cifrecaricate1", solo che nel controllo di cifrecaricate2 se il contenuto del registro $t3 è uguale a 0, si salta alla procedura "secondoPositivo".

```
439 secondoPositivo:
440
441         lw $v0,0($sp)
442         addi $sp,$sp,4
443
444         move $v1,$s2
445
446         j fineparse
```

Viene recuperato il primo operando dallo stack pointer e caricato in $v0, nello stack allora vengono rimossi 4 byte liberi.
Il secondo operando che era contenuto in $s2 viene spostato in $v1 e poi si salta alla procedura "fineparse".

```
448 secondaLettera:
449
450         addi $t0,$t0,2
451         sw $t0,idx
452
453         addi $sp,$sp,-4
454         sw $ra,0($sp)
455
456         lw $t0,idx
457         la $a1,buffer
458         add $a1,$a1,$t0
459         lb $a0,($a1)
460
461         jal switchoperazione
462
463         lw $ra,0($sp)
464         addi $sp,$sp,4
465
466         move $v1,$v0
467
468         lw $v0,0($sp)
469         addi $sp,$sp,4
```

La procedura è simile alla procedura già spiegata in precedenza "primaLettera", la differenza sta però nella parte finale, in primaLettera si salvava tutto nello stack, in questo caso invece vengono fatte altre due operazioni.
Viene spostato il contenuto del registro $v0 in $v1, così il registro $v0 sarà libero per poi inserirci il nuovo primo operando.
Poi si ha il caricamento del primo operando in $v0 precedentemente salvato nello stack e la deallocazione di 4 byte dello stack pointer.

```
471 fineparse:
472         jr $ra
```

In questo caso si salta al registro della funzione chiamante, ovvero si continua la funzione che aveva richiamato il *parse.*

Procedura *ricerca*:

```
476 ricerca:
477         la $t1,buffer
478         lw $a1,idx
479         move $t6,$a0
480         move $t5,$a1
481         add $t1,$t1,$a1
482 loop:
483         lbu $t2,($t1)
484         beq $t2,$t6,trovato
485         beq $t2,$zero,errricerca
486         add $t1,$t1,1
487         addi  $t5, $t5, 1
488         j loop
489 trovato:
490         addi $t5,$t5,1
491         sw $t5, idx
492         jr $ra
```

La procedura *ricerca* carica inizialmente la stringa letta da file in $t1 e carica l'indice idx in $a1 poi sposta il carattere in $t6 (con l'operazione move), sposta l'indice idx in $t5 ed aggiunge l'indice alla stringa per far iniziare la ricerca da quel punto.

Loop è un ciclo che legge ogni carattere della stringa, carica in $t2 il primo byte del buffer e se la lettera è uguale a quella ricercata salto all'etichetta trovato, se è uguale a 0 vuol dire che la stringa invece è finita. A questo punto aumento di 1 l'indice che scorre la stringa, aggiorno anche il contatore delle posizioni già visitate e salto nuovamente all'etichetta *loop*.

La procedura *trovato* aggiunge 1 nel registro $t5, dove vi è registrata la posizione trovata, salva poi il valore ottenuto nel registro di idx e salta al registro del chiamante.

Procedure *stampaIntervallo*:

```
533 stampaIntervallo:
534         la $t1,buffer
535         add $t1, $t1, $a0
536         move $t3,$a0
537         li $v0, 4
538         la $a0, stampafreccie
539         syscall
540 loopt:
541     lb $t2,($t1)
542     beq $t3,$a1,fineStampaIntervallo
543     move $a0,$t2
544     li $v0,11
545     syscall
546     addi $t1,$t1,1
547     addi  $t3, $t3, 1
548     j loopt
549 fineStampaIntervallo:
550         jr $ra
```

La procedura *stampaIntervallo* carica in $t1 il buffer per la stampa e poi aggiunge la posizione iniziale al buffer per far iniziare la stampa da quel punto.
Viene salvata la posizione iniziale in $t3; con una chiamata di sistema con codice di funzione 4 (stampa Stringa) viene stampata la stringa caricata in $a0.

Loopt carica in $t2 il primo byte da stampare e se il contenuto del registro $t3 è uguale alla posizione di arrivo, vado all'etichetta fineStampaIntervallo altrimenti continuo e sposto il carattere in $a0 per essere stampato, infatti si effettua una chiamata di sistema con codice di funzione 11, codice di stampa carattere. Poi viene incrementato il puntatore al buffer ed infine viene incrementato il registro $t3, che contiene il numero di posizioni visitate.

Un salto a *FineStampaIntervallo* invece fa tornare il programma al chiamante.

Procedure *stampaTab*:

```
554 stampaTab:
555         lw $t0,indent
556 loopTab:
557         beq $t0, $zero, fineTab
558         li $a0,9
559         li $v0,11
560         syscall
561         subi $t0,$t0,1
562         j loopTab
563 fineTab:
564         jr $ra
```

La procedura *stampaTab* carica il valore di indent in $t0 e entrando nel ciclo *loopTab,* se $t0 è uguale a 0 salta direttamente a fineTab altrimenti procede caricando il carattere 9 (il carattere che indica un TAB in codice ASCII) in $a0 per essere stampato e con la chiamata di sistema viene stampato il tab. Dopodiché viene sottratto 1 al contatore dei tab, che si trova nel registro $t0 e si salta nuovamente a loopTab.

Saltando all'etichetta *fineTab* si torna alla funzione chiamante.

Procedura *trovafine*:

```
568  trovafine:
569          la $t1,buffer
570          li $t6,'('
571          li $t4,')'
572          lw $t5,idx
573          li $t3,1
574          add $t1,$t1,$t5
```

La procedura *trovafine* carica in $t1 l'indirizzo del buffer che viene utilizzato per la ricerca del carattere. I caratteri da ricercare sono '(' e ')' e vengono caricati rispettivamente in $t6 e $t4.
Si ha poi il caricamento della posizione successiva alla parentesi '(' corrispondente alla funzione in cui ci troviamo, l'inizializzazione del registro $t3 con 1 perché abbiamo già superato la prima parentesi aperta '(', in seguito viene aggiunto idx a $t1 per far puntare $t1 all'idx-esimo carattere di buffer.

```
575  looptf:
576          lb $t2,($t1)
577          beq $t2,$zero,finetrovafine
578          beq $t3,$zero,finetrovafine
579          beq $t2,$t6,apertura
580          beq $t2,$t4,chiusura
581  increase:
582          addi $t1,$t1,1
583          addi  $t5, $t5, 1
584          j looptf
```

La procedura *looptf* carica il primo byte di $t1 in $t2 e se $t2 è uguale a 0 significa che la stringa è finita e si passa alla procedura "finetrovafine", se $t3 è uguale a 0 abbiamo trovato la posizione della parentesi ')' e si passa ugualmente a "finetrovafine".
Se troviamo una '(' saltiamo alla procedura "apertura", se troviamo una ')' saltiamo a "chiusura".

La procedura *increase* aggiunge al registro $s1, in cui era contenuto il buffer, 1 e poi incrementa di uno il registro $t5 che alla fine conterrà il carattere cercato.

```
585  apertura:
586          addi  $t3, $t3, 1
587          j increase
588  chiusura:
589          subi    $t3, $t3, 1
590          j increase
```

La procedura *apertura* incrementa di uno il registro $t3, che conteneva il contatore di parentesi e salta nuovamente alla procedura increase, mentre la procedura *chiusura* decrementa il registro $t3 di 1 e salta poi ad increase.

```
591  finetrovafine:
592          move $v0,$t5
593          jr $ra
```

Viene spostato il registro$t5, che conteneva la posizione del carattere ')', in $v0 e poi si salta al registro della procedura chiamante.

Procedure di errore:

```
596 errfile:
597         li      $v0, 4
598         la      $a0, fnf
599         syscall
600         j fine
601
602 errricerca:
603         li      $v0, 4
604         la      $a0, fnf
605         syscall
606         j fine
607
608 errdivzero:
609         li      $v0, 4
610         la      $a0, divzero
611         syscall
612         j fine
```

Tutte e tre le procedure di errore hanno struttura uguale che permette di stampare una stringa di errore con una chiamata di sistema, per cui appunto si carica il codice 4 nel registro $v0, codice di stampa stringa, e si carica in $a0 il nome della stringa da stampare, inizializzato all'inizio del programma.

Le procedure *errfile* ed *errricerca* stampano la stringa chiamata "fnf": "File non trovato o si è verificato un errore nella lettura di ", mentre *errdivzero* stampa la stringa chiamata "divzero":" Impossibile effettuare una divisione per zero".

Dopo aver eseguito il messaggio di errore il programma non è in grado di proseguire e passa immediatamente alla procedura di fine.

Procedura *fine*:

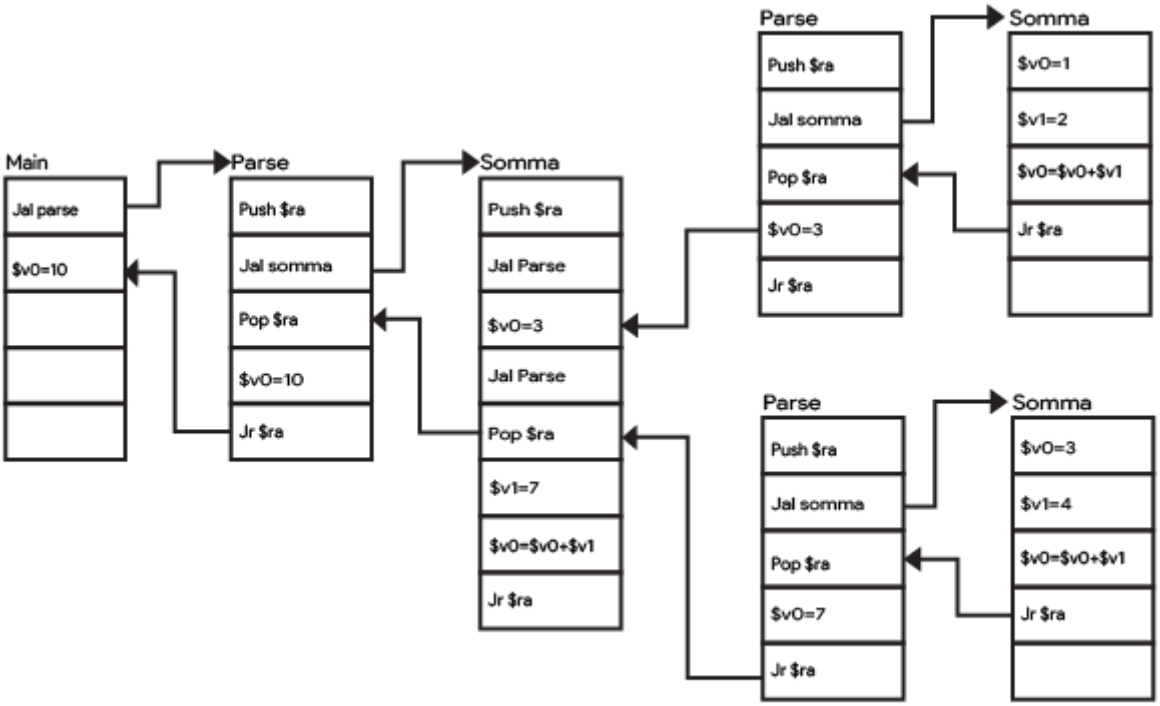
```
614 fine:
615         li      $v0, 10
616         syscall
```

La procedura fine effettua con una chiamata di sistema con codice 10 caricato in $v0, terminando il programma.

Evoluzione dello stack.

In questa immagine sono mostrati i salvataggi del registro $ra nello stack nel caso di input della stringa:

S= “somma(somma(1,2),somma(3,4))”



Il main effettua una chiamata alla funzione Parse, controllando la stringa trova una Somma e richiama la funzione Somma, la funzione somma a sua volta richiama Parse due volte.

Le due funzioni Parse richiamano anch’esse la funzione Somma dopo aver analizzato la stringa e restituiscono i risultati alle procedure chiamanti.

Ad ogni richiamo di una funzione annidata si deve salvare il registro $ra nello stack e prelevarlo dopo la chiamata.

SCHEDULER DI PROCESSI.

Esercizio del progetto numero 2

Data consegna: 03 luglio 2016

Autori:

Salani Lorenzo: lorenzo.salani@stud.unifi.it

Fagioli Giulio: giulio.fagioli@stud.unifi.it

Buonanno Cecilia: cecilia.buonanno@stud.unifi.it

Sommario

Descrizione dell'esercizio.

Il testo dell'esercizio richiedeva di scrivere un programma in assembly MIPS, che simuli la gestione dei task (o processi) di un sistema operativo.

Un task è definito dalle seguenti informazioni:
-un identificatore numerico (ID),
-un nome (NOME TASK, stringa che può contenere al massimo 8 caratteri)
-una priorità (PRIORITA' numero compreso fra 0 e 9)
-il numero di cicli di esecuzione che richiede per essere completato (ESECUZIONI RIMANENTI massimo 99 cicli)

Lo scheduler è una funzionalità del sistema operativo che gestisce una coda in cui i task
vengono mantenuti ordinati in base alla politica di scheduling adottata. In questo esercizio
supporremo l'esistenza di due possibili politiche di scheduling:
• a) Scheduling su PRIORITA' (scheduling di default); si ordinano i task in modo decrescente rispetto al valore di PRIORITA' ad essi associato (quindi da 9 a 0). Il task da eseguire per primo sarà quello a cui è stato associato il numero di PRIORITA' più basso.
• b) Scheduling su ESECUZIONI RIMANENTI; si ordinano i task in modo crescente rispetto al numero di ESECUZIONI RIMANENTI necessarie per il loro completamento. Il task da eseguire per primo sarà quello che necessita del maggior numero di ESECUZIONI RIMANENTI per essere completato.

I task con uguale valore di PRIORITA' o di ESECUZIONI RIMANENTI dovranno essere ordinati in modo decrescente rispetto al loro identificatore numerico.

Le operazioni principali che devono essere implementate sono le seguenti:

1. Inserire un nuovo task, richiedendo tutti i campi del record. Il task inserito dovrà poi essere posizionato nella coda a seconda della politica di scheduling utilizzata.
2. Eseguire il task che si trova in testa alla coda;
3. Eseguire il task il cui identificatore ID è specificato dall'utente;
4. Eliminare il task il cui identificatore ID è specificato dall'utente;
5. Modificare la PRIORITA' del task il cui identificatore ID è specificato dall'utente;
6. Cambiare la politica di scheduling utilizzata, passando dalla a) alla b) e viceversa;
7. Uscire dal programma.

Soluzione adottata.

In questo capitolo faremo riferimento unicamente alla modalità utilizzata per la risoluzione di tale esercizio in linguaggio naturale.

Un task viene memorizzato dalle seguenti informazioni:
-ID: identificatore numerico (4 byte)
-NOME TASK: stringa che può contenere al massimo 8 caratteri (8 byte)
-PRIORITA: numero compreso fra 0 e 9 (4 byte)
-ESECUZIONI RIMANENTI: Il numero di cicli di esecuzione che richiede per essere completato (4 byte)

Aggiungendo ai campi di un task il puntatore all'elemento precedente e il puntatore all'elemento successivo, in totale un task occupa 28 byte, i quali vendono allocati nell'heap ad ogni inserimento.

La nostra idea è stata quella di gestire la scelta dell'operazione da fare tramite uno "switch" implementato con una jump table, cioè una tabella dove vengono memorizzati tutti gli indirizzi delle 7 procedure da richiamare.
Ogni procedura esegue il compito precedentemente descritto ed al termine dell'esecuzione di una di esse viene effettuato l'ordinamento in base alla politica di scheduling adottata.

Per eseguire l'ordinamento decrescente rispetto all'ID dei task, con uguale valore di PRIORITA' o di ESECUZIONI RIMANENTI, si utilizza una procedura stabile di ordinamento.
La procedura di ordinamento utilizzata è un Selection Sort reso stabile effettuando gli scambi dei valori adiacenti al posto di scambiarli direttamente.
Un ordinamento stabile è un ordinamento che mantiene l'ordine relativo delle chiavi uguali, quindi basterà effettuare prima un ordinamento rispetto all'ID e successivamente uno in base alla politica di scheduling utilizzata.

I registri $t8 e $t9 in questo programma sono utilizzati unicamente come puntatori al primo elemento (coda) ed all'ultimo elemento (testa) memorizzato nell' heap.

Sono presenti funzioni che hanno compiti molto diversi e più complicati, i quali spiegheremo più approfonditamente nel prossimo capitolo, analizzando anche il codice con i rispettivi commenti.

Analisi del codice.

In questo capitolo verrà analizzato tutto il codice in assembly, le varie procedure, i registri, le variabili utilizzate e tutte le operazioni in modo dettagliato.

Il seguente codice è l'inizio del nostro progetto, in esso vengono dichiarate le variabili globali:

```
1  .data
2          jump_table: .space 28
3          schedulingType : .word 0
4          id:         .word 0
5
```

La *jump_table* è un'area di memoria che conterrà 7 word (28 bytes), instanziata nel main con gli indirizzi delle label che richiameranno le procedure corrisponenti al menu principale.

La word *schedulingType* contiene il valore corrispondente al tipo di scheduling attualmente utilizzato, 0 per lo scheduling su priorità, 1 per lo scheduling su esecuzioni rimanenti, esso viene inizializzato a 0 impostando come scheduling predefinito quello su priorità.

La word *Id* viene inizializzata con 0 ed è l'Id da assegnare al prossimo task inserito.

```
6  #Stringhe
7  #Menu
8  stringaBenvenuto: .asciiz "Benvenuto nel simulatore di scheduler di processi.\n"
9  stringaMenu : .asciiz "1) Inserisci un nuovo Task\n2) Esegui primo task in testa alla coda\n3) Esegui il task con id
10 inserimento:.asciiz "Inserisci un numero da 1 a 7\n"
11 #Inserimento
12 inserisciPriorita:    .asciiz "Inserisci la priorita' del task (fra 0 e 9) \n"
13 inserisciNome:    .asciiz "Inserisci il nome del task (max 8 caratteri)\n"
14 inserisciEsecuzione:    .asciiz "Inserisci le esecuzioni necessarie per il completamento del task (fra 1 e 99)\n"
15 #Esecuzione task
16 taskRimosso: .asciiz "Esecuzioni rimanenti 0, il task e' stato rimosso.\n"
17 #Cambia scheduling
18 politicaUtilizzata: .asciiz "Politica utilizzata:"
19 schedulingPriorita:    .asciiz "Scheduling su Priorita'\n"
20 schedulingEsecuzioni:   .asciiz "Scheduling su Esecuzioni rimanenti\n"
21 cambiaScheduling:          .asciiz "Vuoi cambiare la politica di scheduling utilizzata? (y/n) \n"
22 politicaCambiata : .asciiz "\nPolitica cambiata.\n"
23 #Ricerca id
24 inserisciId : .asciiz "Inserisci l'id:"
25 idNonTrovato: .asciiz "Id non trovato.\n"
26 #Modifica priorita'
27 priorita: .asciiz "Priorita': "
28 cambiaPriorita: .asciiz "Inserisci la nuova priorita': "
29 #Varie
30 operazioneNonConsentita:   .asciiz "Operazione consentita con almeno un task nella coda.\n"
31 operazioneOk:   .asciiz "Operazione effettuata con successo.\n"
32 erroreScelta:.asciiz "Il numero inserito non era compreso tra 1 e 7\n"
33 fine:.asciiz "Termine simulatore\n\n"
34 #Stringhe tabella
35 spazio:    .asciiz " "
36 stanga: .asciiz "|"
37 stangaFineTab: .asciiz "|\n"
38 inizioStringaDati: .asciiz "|   "
39 fineStringaDati: .asciiz "  |"
40 fineStringaDatiDoppia: .asciiz "   |"
41 primaLineaTab: .asciiz "+------+------------+-------------+-----------------------+\n|  ID  |  PRIORITA' |  NOME TASK
42 ultimaLienaTab: .asciiz "+------+------------+-------------+-----------------------+\n"
```

Il frammento di codice dalla riga 5 alla riga 42 contiene tutte le stringhe che verranno stampate quando richiamate nel codice con la Syscall 4.

Procedura *"main"*:

```
.globl main
main:
# Prepara la jump_table con gli indirizzi delle case actions
        la $t1, jump_table
        la $t0, inserimentoTask
        sw $t0, 0($t1)
        la $t0, eseguiTask
        sw $t0, 4($t1)
        la $t0, eseguiTaskId
        sw $t0, 8($t1)
        la $t0, rimuoviTaskById
        sw $t0, 12($t1)
        la $t0, modificaPriorita
        sw $t0, 16($t1)
        la $t0, modificaScheduling
        sw $t0, 20($t1)
        la $t0, exit
        sw $t0, 24($t1)

        la $a0,stringaBenvenuto
        li $v0,4
        syscall

printMenu:
        la $a0,stringaMenu
        li $v0,4
        syscall
# Inserimento del numero per effettuare la scelta
sceltaMenu:
        li $v0,4
        la $a0, inserimento
        syscall

        li $v0, 5
        syscall

        move $t2, $v0
```

```
# Inserimento del numero per effettuare la scelta
sceltaMenu:
        li $v0,4
        la $a0, inserimento
        syscall

        li $v0, 5
        syscall

        move $t2, $v0

        sle  $t0, $t2, $zero
        bne  $t0, $zero, errScelta
        li   $t0,7
        sle   $t0, $t2, $t0
        beq  $t0, $zero, errScelta
# Gestione della scelta
branch_case:
        blt $t2,2,continuaBranch
        bgt $t2,5,continuaBranch

        bne $t8,$t9,continuaBranch
        bne $t8,$zero,continuaBranch

        li $v0,4
        la $a0, operazioneNonConsentita
        syscall

        j sceltaMenu

continuaBranch:

        la $t1, jump_table
```

La procedura main è la procedura principale del programma in cui vengono richiamate tutte le altre procedure, la prima operazione effettuata dal main è quella di salvare gli indirizzi delle etichette nella jump_table, le quali indicano l'inizio di una procedura, vengono usate le istruzioni 'load address' e 'save word', salvando 4 byte alla volta (1 word).

Completata l'inizializzazione della jump table stampiamo sullo schermo il primo messaggio di benvenuto e subito dopo entriamo nella prima funzione printMenu che si occupa di stampare il menù delle possibili scelte che l'utente può effettuare.

Finita la funzione addetta alla stampa del menu iniziale si entra nella seconda funzione, in essa verrà richiesto all'utente quale delle 7 operazioni consentite vorrà effettuare.

Questa operazione viene fatta leggendo un numero intero compreso fra 1 e 7, vengono gestiti eventuali errori come l'inserimento errato del numero, fino a che l'utente uscirà dal programma con l'inserimento del numero 7.

Per far ciò sfruttiamo l'istruzione sle, la quale controlla se il secondo registro è minore del terzo e in questo caso mette nel primo registro 1 altrimenti 0.

Questa operazione viene fatta due volte salvando nel terzo registro prima 0 e dopo 7, essendo così sicuri che il numero inserito sia compreso fra quest'ultimi due.

La prossima funzione da eseguire sarà branch_case.

```
continuaBranch:

        la $t1, jump_table
        addi $t2, $t2, -1
        add $t0, $t2, $t2
        add $t0, $t0, $t0
        add $t0, $t0, $t1
        lw $t0, 0($t0)
        jalr $t0

        beq $t8,$t9,stampaRecord

        jal sortById

        lw $t0, schedulingType

        beq $t0,1,ordinamentoEsecuzioni
        jal sortByPriorita

        j stampaRecord

        ordinamentoEsecuzioni:
        jal sortByEsecuzioni

        stampaRecord:
        bne $t8,$t9,continuaStampa
        bne $t8,$zero,continuaStampa
        j printMenu
        continuaStampa:
        jal stampaTabella

        j printMenu
```

La funzione branch_case si occupa di “convertire” la scelta dell'utente nell'operazione corrispondente e di mantenere ordinata la coda affinché le operazioni possano essere effettuate correttamente in ogni momento; inizialmente vengono effettuati alcuni controlli per assicurarsi che l'utente non possa eseguire operazioni su un insieme vuoto di task, in tal caso verrà stampato un messaggio di errore e l'utente sarà riportato al menù iniziale dove potrà scegliere nuovamente l'operazione da eseguire.

Se tutti i controlli hanno dato esito positivo saltiamo all’etichetta continuaBranch.

Saltando a continuaBranch la jump table sarà caricata nel registro $t1 e il numero inserito dall'utente sarà diminuito di uno poiché in memoria l'intervallo sarà da 0 a 6, questo numero viene moltiplicato prima per 4, in fine è aggiunto all'indirizzo di inizio della jump table puntando così all'indirizzo in cui è stata memorizzata la label corrispondente all'operazione scelta dall'utente; infine con l’istruzione jalr (jump and link register) si richiama la procedura richiesta.

Viene eseguito un controllo sulla coda contenente i task al fine di non dover compiere operazioni inutili come l'ordinamento dei dati nel caso in cui non ci sia elementi nella coda o ce ne sia solo 1, in questo caso salteremo direttamente alla stampa della tabella, altrimenti si eseguirà un ordinamento usando come chiave l'Id del task richiamando la funzione sortById.

Caricheremo in memoria il tipo di scheduling e lo controlleremo, nel caso fosse 1 andremo ad ordinare i dati presenti nella struttura usando come chiave le esecuzioni rimanenti dei task richiamando sortByEsecuzioni o nel caso fosse 0 l'ordinamento verrà effettuato usando come chiave la priorità richiamando sortByPriorita.

Verrà stampato il task dopo aver eseguito due controlli, se la testa e la coda della struttura sono uguali fra loro e a 0 non verrà stampata a schermo la tabella non essendoci task da stampare.

Alla fine della procedura si effettua un salto all’etichetta printMenu per richiedere nuovamente una operazione da effettuare.

I controlli sulla coda sono effettuati nella funzione main e vengono richiamate le altre funzioni in base al numero di elementi nella coda, quindi richiamando la funzione eseguiTask siamo sicuri che ci sia almeno un task nella coda.

Procedura *"InserimentoTask":*

```
#Case 1 : Funzione che permette di inserire un task
inserimentoTask:
# Allocazione di 28 byte per il record da inserire
        li $v0, 9
        li $a0, 28
        syscall
        move $s0, $v0
# Inserimenti
        sw $zero, 0($s0)
# Inserimento id
        lw $t0, id
        sw $t0, 4($s0)

        addi $t0, $t0, 1
        sw $t0, id
# Inserimento nome
        li $v0,4
        la $a0, inserisciNome
        syscall

        li $a1, 9
        la $a0, 8($s0)
        li $v0, 8
        syscall

# Inserimento priorita'
        li $v0,4
        la $a0, inserisciPriorita
        syscall

reinserisciPriorita:
        li $v0, 5
        syscall

        bgt $v0,9, reinserisciPriorita
        blt $v0,0, reinserisciPriorita

        sw $v0, 16($s0)
```

Passiamo ora alle operazioni che l'utente può effettuare, la prima che troviamo è l'inserimento di un task e corrisponde al case 1 della jump_table.

Verrà allocata subito la quantità di memoria necessaria per memorizzare tutte le informazioni relative al task, nel nostro caso 28 byte, poiché memorizzeremo due puntatori, uno per il task precedente e uno per quello successivo ad ogni record.

Dopo aver spostato l'indirizzo della memoria appena allocata in $s0, il puntatore all'elemento precedente (0($s0)) viene settato a 0 e viene memorizzato nel campo Id (4($s0)) il contenuto della word 'Id' in cui salviamo ad ogni passaggio l'Id di un possibile nuovo task, aggiorniamo quindi il valore memorizzato nella word 'Id' con il suo successivo.

Verrà chiesto all'utente di immettere le tre informazioni del task ovvero: nome (con una lunghezza minima di 1 e massima di 8), priorità (valore compreso fra 0 e 9) e esecuzioni rimanenti (valore compreso fra 1 e 99).

Per l'inserimento del nome (8($s0)) permetteremo all'utente di inserire al massimo 8 caratteri (2 word). Questo ci permette di evitare il controllo sulla quantità di caratteri inseriti in un secondo momento.

```
reinserisciPriorita:
        li $v0, 5
        syscall

        bgt $v0,9, reinserisciPriorita
        blt $v0,0, reinserisciPriorita

        sw $v0, 16($s0)

#Esecuzioni
        li $v0,4
        la $a0, inserisciEsecuzione
        syscall

reinserisciEsecuzione:
        li $v0, 5
        syscall
        bgt $v0,99, reinserisciEsecuzione
        blt $v0,1, reinserisciEsecuzione

        sw $v0, 20($s0)

        sw $zero, 24($s0)
# Modifica puntatori record successivo e precedente
        bne $t8, $zero, linkLast
        move $t8, $s0
        move $t9, $s0
        j fineInserimento
linkLast:
        sw $s0, 24($t9)
        sw $t9, 0($s0)
        move $t9, $s0
fineInserimento:
        li $v0,4
        la $a0, operazioneOk
        syscall

        jr $ra
```

Si passerà poi all'inserimento della priorità (16($s0)) effettuando un controllo per assicurarsi che il numero inserito sia compreso realmente nell'intervallo prestabilito, quindi nel caso sia minore di 0 o maggiore di 9, l'utente dovrà reinserire la priorità, se invece appartiene all'intervallo essa verrà memorizzata.

L'ultima informazione da inserire sono le esecuzioni rimanenti (20($s0)), stampando la modalità di inserimento e controllando che il valore inserito rientri nell'intervallo scelto, dopo l'inserimento delle esecuzioni si imposterà il campo del puntatore all'elemento successivo (24($s0)) a 0.

L'ultima parte della funzione di inserimento è quella che si occupa della gestione della testa e della coda, se la coda è vuota impostiamo i puntatori alla testa ($t9) e alla coda ($t8) uguali ad $s0 che conteneva il puntatore all'elemento appena inserito.

Nel caso in cui la coda non fosse vuota verrà effettuato un salto all'etichetta linklast nella quale verrà memorizzato l'indirizzo del nuovo record nell'ultimo campo dell'ultimo record, l'indirizzo dell'ultimo record nel primo campo del nuovo record e infine l'ultimo record prende l'indirizzo del nuovo record.

L'esecuzione della funzione verrà terminata stampando una stringa che confermerà la corretta immissione del task e la jr $ra ci permetterà di tornare indietro alla branch_case.

Procedura *"eseguiTask"*:

```
209  #Case 2: Funzione di esecuzione del task in testa
210  eseguiTask:
211
212          lw $t1, 20($t9)
213          addi $t1,$t1,-1
214
215          bne $t1, $zero, mantieniTask
216  # Se il numero di esecuzioni rimanenti e' 0 dopo l'ultima esecuzione rimuovo il task
217          addi $sp,$sp,-4
218          sw $ra,0($sp)
219
220          move $a0, $t9
221          jal rimuoviTask
222
223          lw $ra,0($sp)
224          addi $sp,$sp,4
225
226          li $v0,4
227          la $a0, taskRimosso
228          syscall
229
230          jr $ra
231
232  mantieniTask:
233          sw $t1, 20($t9)
234
235          li $v0,4
236          la $a0, operazioneOk
237          syscall
238
239          jr $ra
```

L'esecuzione del task in testa alla coda è l'operazione che l'utente può effettuare inserendo '2' nel menu principale.

L'esecuzione di un task consiste nel salvare il numero di esecuzioni rimanenti (20($t9)) del task in testa e decrementarlo di uno, dopo di che si controlla se esso è uguale a 0.

Nel caso in cui sia diverso da 0 saltiamo all'etichetta mantieniTask e dopo aver salvato il nuovo valore di esecuzioni rimanenti del task e stampato il messaggio di corretta esecuzione della funzione, torniamo al chiamante.

Se il numero di esecuzioni rimanenti già decrementato di uno è uguale a 0, salviamo il registro $ra nello stack perché dopo l'esecuzione della funzione successiva tramite una jal sovrascriverebbe il valore di $ra.

La chiamata a rimuoviTask impostando l'argomento $a0 uguale al puntatore alla testa eseguirà l'eliminazione del task, dopo di che viene ripristinato il valore di $ra e stampato su schermo un messaggio di avvenuta rimozione del task a causa di un numero di esecuzioni rimanenti pari a 0.

Procedura *“eseguiTaskById”*:

```
242  #Case 3
243  eseguiTaskId:
244
245          addi $sp,$sp,-4
246          sw $ra,0($sp)
247
248          li $v0,4
249          la $a0, inserisciId
250          syscall
251
252          li $v0, 5
253          syscall
254
255          move $a0,$v0
256          jal ricercaId
257
258          beq $v0,-1, errIdRicercaNonTrovato
259
260          move $t0,$v0
261          lw $t1, 20($t0)
262          addi $t1,$t1,-1
263
264          bne $t1, $zero, mantieniTaskId
265          move $a0, $t0
266          jal rimuoviTask
267          lw $ra,0($sp)
268          addi $sp,$sp,4
269
270          jr $ra
271  mantieniTaskId:
272          sw $t1, 20($t0)
273          lw $ra,0($sp)
274          addi $sp,$sp,4
275
276          li $v0,4
277          la $a0, operazioneOk
278          syscall
279
280          jr $ra
```

La terza operazione che possiamo eseguire nel menu principale è l’eliminazione di un task inserendo il suo Id.

Per far ciò verrà stampata una stringa che chiederà l’inserimento di un Id identificativo ed usando la syscall 5 si permetterà all’utente di inserire un numero da tastiera.

Dopo aver caricato il valore appena inserito dall'utente in $a0 per essere passato come parametro, useremo la funzione ricercaId per ricercare un task contenente l'Id selezionato, prima però come nel caso precedente salviamo l'indirizzo contenuto nel registro $ra per poterlo ripristinare dopo.

La funzione ricercaId restituirà come valore di ritorno nel registro $v0 il valore -1 se non è stato trovato nessun task nella lista con l'Id specificato oppure l'indirizzo corrispondente al record con l'Id cercato.

Nel primo caso saltiamo all’etichetta errRicercaIdNonTrovato, mentre nel secondo caso il programma si comporta esattamente come nella funzione eseguiTask, cioè rimuoverà il task se ha 0 esecuzioni rimanenti oppure decrementerà il valore delle esecuzioni rimanenti del task e prima di ritornare al chiamante verrà ripristinato $ra dallo stack.

```
282  errIdRicercaNonTrovato:
283          lw $ra,0($sp)
284          addi $sp,$sp,4
285
286          li $v0,4
287          la $a0, idNonTrovato
288          syscall
289          jr $ra
```

Se l’Id non è stato trovato dalla procedura ricercaId il programma eseguirà errRicercaIdNonTrovato, che dopo aver ripristinato $ra dallo stack, stampa la stringa che informa l’utente dell’avvenuto inserimento di un Id non presente nella coda, successivamente ritorniamo alla procedura chiamante.

Procedura *"rimuoviTaskById"*:

```
291  #Case 4
292  rimuoviTaskById:
293
294          li $v0,4
295          la $a0, inserisciId
296          syscall
297
298          li $v0, 5
299          syscall
300
301          addi $sp,$sp,-4
302          sw $ra,0($sp)
303
304          move $a0,$v0
305          jal ricercaId
306
307          beq $v0,-1, errIdRicercaNonTrovato
308
309          move $a0,$v0
310          jal rimuoviTask
311
312          lw $ra,0($sp)
313          addi $sp,$sp,4
314
315          li $v0,4
316          la $a0, operazioneOk
317          syscall
318
319          jr $ra
320
```

Inserendo '4' nel menu principale l'utente avrà la possibilità di rimuovere un task specificando il suo Id tramite la procedura rimuoviTask.

Inizialmente la procedura stamperà a schermo la stringa di richiesta dell'inserimento di un task e tramite la syscall 5 permetterà all'utente di inserire l'Id numerico che identifica il task da eliminare.

Dopo aver salvato il registro $ra nello stack salveremo l'Id inserito dall'utente in $a0 per passarlo come parametro alla funzione ricercaId richiamata al rigo successivo

Il valore di ritorno della funzione sarà l'indirizzo del task il cui Id è stato passato come parametro, se questo valore è -1 saltiamo all'etichetta errRicercaIdNonTrovato perché è stato inserito un Id non presente nella coda.

Se l'Id inserito è presente nella coda e quindi la funzione ha restituito l'indirizzo di esso, lo salviamo in $a0 per essere passato come parametro alla funzione rimuoviTask, che rimuoverà il task dalla coda.

Dopo aver ripristinato il vecchio valore di $ra, stampiamo a video la stringa di corretta rimozione del task e tramite l'istruzione jr torniamo alla procedura chiamante.

```
282  errIdRicercaNonTrovato:
283          lw $ra,0($sp)
284          addi $sp,$sp,4
285
286          li $v0,4
287          la $a0, idNonTrovato
288          syscall
289          jr $ra
```

Come nel caso precedente se l'Id non è stato trovato dalla procedura ricercaId il programma eseguirà errRicercaIdNonTrovato, che dopo aver ripristinato $ra dallo stack, stampa la stringa che informa l'utente dell'avvenuto inserimento di un Id non presente nella coda, successivamente ritorniamo alla procedura chiamante.

Procedura *"modificaPriorita"*:

```
321 #Case 5
322 #Funzione che richiede l'id e permette la modifica della priorita del task
323 modificaPriorita:
324         li $v0,4
325         la $a0, inserisciId
326         syscall
327
328         li $v0, 5
329         syscall
330
331         addi $sp,$sp,-4
332         sw $ra,0($sp)
333
334         move $a0,$v0
335         jal ricercaId
336
337         lw $ra,0($sp)
338         addi $sp,$sp,4
339
340         beq $v0,-1, errIdRicercaNonTrovato
341         move $t0,$v0
342
343         li $v0,4
344         la $a0, cambiaPriorita
345         syscall
346 rimodificaPriorita:
347         li $v0, 5
348         syscall
349
350         bgt $v0,9, reinserisciPriorita
351         blt $v0,0, reinserisciPriorita
352
353         sw $v0, 16($t0)
354
355         li $v0,4
356         la $a0, operazioneOk
357         syscall
358
359         jr $ra
```

La quinta operazione eseguibile nel menu è la modifica della priorità di un task inserendo il suo Id.

Inizialmente la procedura stamperà a schermo la stringa di richiesta dell'inserimento di un task e tramite la syscall 5 permetterà all'utente di inserire l'Id numerico che identifica il task di cui modificarne la priorità.

Dopo aver salvato il registro $ra nello stack, lo passeremo come parametro alla funzione ricercaId, salvando in $a0 l'Id appena inserito dall'utente.

Questa funzione restituirà l'indirizzo del task ricercato oppure in caso di Id non trovato restituirà il valore -1, dopo aver ripristinato il registro $ra prelevandolo dallo stack, controlleremo se l'Id è stato trovato ed in caso contrario salteremo all'etichetta errIdRicercaNonTrovato.

Se l'Id ricercato viene trovato dalla procedura stamperemo la stringa relativa al cambio di priorità e richiederemo all'utente l'inserimento di una nuova priorità per quel task utilizzando la syscall 5.

Come per l'inserimento di un nuovo task sono effettuati i controlli sul valore di priorità inserito, se la priorità inserita non è compresa fra 0 e 9 richiederemo di nuovo l'inserimento di essa, dopo di che la nuova priorità verrà salvata nel quinto campo del record e ritorneremo alla procedura chiamante.

```
282 errIdRicercaNonTrovato:
283         lw $ra,0($sp)
284         addi $sp,$sp,4
285
286         li $v0,4
287         la $a0, idNonTrovato
288         syscall
289         jr $ra
```

Come nel caso precedente se l'Id non è stato trovato dalla procedura ricercaId il programma eseguirà errRicercaIdNonTrovato, che dopo aver ripristinato $ra dallo stack, stampa la stringa che informa l'utente dell'avvenuto inserimento di un Id non presente nella coda, successivamente ritorniamo alla procedura chiamante.

Procedura *"modificaScheduling"*:

```
361  #Case 6
362  modificaScheduling:
363          li $v0,4
364          la $a0, politicaUtilizzata
365          syscall
366          lw $t0, schedulingType
367          beq $t0, 0, schedPriorita
368
369          li $v0,4
370          la $a0, schedulingEsecuzioni
371          syscall
372          j continuaScheduling
373  schedPriorita:
374          li $v0,4
375          la $a0, schedulingPriorita
376          syscall
377  continuaScheduling:
378          li $v0,4
379          la $a0, cambiaScheduling
380          syscall
381
382          li $v0, 12
383          syscall
384
385          beq $v0,'y',cambio
386          j fineCambio
```

```
387  cambio:
388          beq $t0,0,setTo1
389          li $t0,0
390          sw $t0,schedulingType
391          j fineCambio
392  setTo1:
393          li $t0,1
394          sw $t0,schedulingType
395          li $v0,4
396          la $a0, politicaCambiata
397          syscall
398  fineCambio:
399          jr $ra
```

La procedura di modifica della politica di scheduling è richiamata dal menu principale inserendo il valore '6' e permette di modificare la politica di scheduling passando da quella per priorità a quella per esecuzioni rimanenti, il passaggio dall'una all'altra determina la scelta dell'ordinamento da fare nel main.

Appena richiamiamo questa procedura essa stamperà la politica in uso, caricando la word schedulingType in $t0 e controllando il suo valore.

Dopo la stampa della politica in uso a partire dall'etichetta continuaScheduling verrà stampata la stringa che chiede se vogliamo cambiare la politica in uso e verrà richiesto l'inserimento di 'y' o 'n', in base alla scelta dell'utente cambiamo o meno la politica di scheduling.

Se la lettera inserita è una 'y' controlliamo in valore di schedulingType e lo cambiamo notificando il cambiamento avvenuto stampando la stringa relativa.

Procedura *"exit"*:

```
402  #Case 7
403  exit: # stampa messaggio di uscita e esce
404          li $v0,4
405          la $a0, fine
406          syscall
407
408          li $v0, 10
409          syscall
```

La procedura richiamata con l'inserimento del numero 7 nel menu stampa a video la stringa di terminazione del programma e tramite la syscall 10 termina il programma.

Procedura *"ricercaId"*:

```
412  #Funzione che ricerca un id e lo restituisce, se non lo trova restituisce -1
413  #Parametro $a0 id da ricercare
414  #Valore di ritorno: $v0, indirizzo del record con l'id cercato
415  ricercaId:
416          move $t0,$t8
417  loopRicercaId:
418          lw $t1, 4($t0)
419          beq $t1, $a0, fineRicerca
420
421          lw $t2, 24($t0)
422          beq $t2, $zero, fineIdNonTrovato
423
424          lw $t0, 24($t0)
425          j loopRicercaId
426  fineRicerca:
427          move $v0, $t0
428          jr $ra
429  fineIdNonTrovato:
430          li $v0,-1
431          jr $ra
```

La procedura ricercaId è utilizzata dalle funzioni che richiedono l'inserimento di un Id, infatti viene richiamata passandogli come parametro un Id in $a0 che verrà ricercato fra gli elementi della coda restituendo l'indirizzo del task con quell'Id, in caso non venga trovato verrà restituito il valore -1.

La procedura inizialmente carica l'indirizzo del primo task della coda in $t0, il ciclo successivo ad ogni passaggio carica l'Id del task (4($t0)) in $t1 e lo confronta con il parametro, se l'Id è stato trovato saltiamo all'etichetta fineRicerca, altrimenti si continua controllando che il puntatore all'elemento successivo non sia 0, in quel caso abbiamo raggiunto la fine della coda.

Alla fine di ogni ciclo facciamo puntare $t0 all'elemento successivo scorrendo cosi la coda fino all'ultimo elemento, nel caso in cui l'Id venga trovato carichiamo l'indirizzo del task con quell'Id in $v0, altrimenti carichiamo -1, dopo di che torniamo alla funzione chiamante.

Procedura *"rimuoviTask"*:

```
#Funzione rimuove il task corrispondente all'indirizzo passato come parametro
#Parametro $a0 indirizzo da eliminare rimuoviTask:
        move $t0,$a0
#Copio il record primo nel record da eliminare
        lw  $t1, 4($t8)
        sw $t1, 4($t0)
        lw  $t1, 12($t8)
        sw $t1, 12($t0)
        lw  $t1, 16($t8)
        sw $t1, 16($t0)
        lw  $t1, 20($t8)
        sw $t1, 20($t0)
#azzerramento del campi del record
        li  $t1, 0
        move $t2, $t8

        beq $t8, $t9, nessunElemento
        lw  $t8, 24($t8)
        sw  $t1, 0($t8)

        j piuElementi
nessunElemento:
        move $t8,$zero
        move $t9,$zero

piuElementi:
        sw $t1, 0($t2)
        sw $t1, 4($t2)
        sw $t1, 8($t2)
        sw $t1, 12($t2)
        sw $t1, 16($t2)
        sw $t1, 20($t2)
        sw $t1, 24($t2)

        jr $ra
```

La procedura rimuoviTask è utilizzata dalle procedure che richiedono l'eliminazione di un task, essa richiede come parametro in $a0 l'indirizzo di un task e lo elimina.

L'eliminazione di un task è effettuata copiando il task in coda nel task da eliminare e dopo azzerando i campi del task in coda.

Inizialmente viene caricato in $t0 l'indirizzo del record da cancellare passato come parametro, dopo vengono copiati i 4 campi del record in coda (senza contare i puntatori) in quelli del record da eliminare mediante 4 operazioni di load word e store word.

Viene caricato in $t1 uno 0 per azzerare i campi e controllato se il puntatore alla coda e alla testa hanno lo stesso valore, in questo caso significa che cera un solo elemento nella coda e che lo dobbiamo eliminare, quindi saltiamo all'etichetta nessunElemento ed azzeriamo i puntatori alla coda e alla testa e tutti i campi dell'elemento.

Nel caso in cui ci sia più elementi modifichiamo il puntatore al fondo della coda con il valore dell'indirizzo del suo successivo (24($t8)), dopo di che saltando a piuElementi azzeriamo i campi puntati dal vecchio valore del puntatore alla coda precedentemente salvato in $t2.

Quando l'elemento è stato azzerato ritorniamo alla procedura chiamante.

Procedure di ordinamento.

Procedura *"sortById"*:

```
474  #Sort decrescente per id
475  sortById:
476  ricerca:
477          move $t0, $t8
478  #Ricerca del massimo fra $t0 e la fine della coda
479  ricercaMassimo:
480          lw $t1, 24($t0)
481          beq $t1, $zero, fineRicercaMassimo
482          lw $t2, 4($t0)
483          move $t4,$t0
484  loopRicerca:
485          beq $t1, $zero, scambi
486          lw $t3, 4($t1)
487          bge $t2, $t3,continua
488          move $t2,$t3
489          move $t4,$t1
490  continua:
491          lw $t1, 24($t1)
492          j loopRicerca
493  scambi:
494  #Quando abbiamo trovato il valore massimo(puntato da $t4)
495  #si effettuano gli scambi degli elementi adiacenti per portarlo nella posizione puntata da $t0
496          lw $t5, 0($t4)
497          bgt $t0, $t5, esci
498
499  scambiaAdiacenti:
500          lw $t6, 4($t4)
501          lw $t7, 4($t5)
502          beq $t6, $t7, nonScambiare
503
504          lw  $t6, 4($t5)
505          lw $t7, 4($t4)
506          sw $t6, 4($t4)
507          sw $t7, 4($t5)
508
509          lw  $t6, 8($t5)
510          lw $t7, 8($t4)
511          sw $t6, 8($t4)
512          sw $t7, 8($t5)
528
529  nonScambiare:
530          lw $t4, 0($t4)
531          lw $t5, 0($t4)
532          bgt $t0,$t5, esci
533          j scambiaAdiacenti
534
535  esci:
536          lw $t0, 24($t0)
537          j ricercaMassimo
538
539  fineRicercaMassimo:
540          jr $ra
```

Il problema di ordinare i dati in base allo scheduling scelto ed ordinare i dati con lo stesso id in ordine decrescente è stato risolto creando 3 procedure di ordinamento.

La procedura di ordinamento per Id ordina i task nell'heap in ordine descrescente cercando ad ogni passaggio il task con Id maggiore e spostandolo nel primo valore ancora da ordinare.

Questa procedura è divisa in due parti, la prima inizia all'etichetta 'Ricerca:' e permette di trovare il task con l'Id più grande; la seconda inizia all'etichetta 'Scambi:' e permette, tramite scambi di elementi adiacenti di spostare il task nella posizione che gli spetta nella coda ordinata.

Questa procedura di ordinamento dopo aver salvato l'indirizzo della coda in $t0 entra in un ciclo eseguito un numero di volte quanti sono i task nella coda; inizialmente si eseguirà la ricerca dell'elemento con Id massimo avvalendosi dei registri $t1, $t2 e $t3, alla riga 485 abbiamo il controllo principale che permette di confrontare due elementi, salvando il massimo ($t2) ed il suo indirizzo ($t4).

Quando raggiungiamo la fine della coda saltiamo alla procedura 'Scambi:', la procedura che scambia gli elementi adiacenti portando il maggiore, il cui indirizzo è salvato in $t4, nella posizione puntata da $t0. Nel caso in cui i due task da scambiare avessero lo stesso Id non si effettua lo scambio (riga 500-501); alla fine dello scambio facciamo puntare $t0 al task successivo e eseguiamo nuovamente le procedure di ricerca del minimo e scambi fino ad avere una coda ordinata rispetto all' Id.

Le due procedure seguenti verranno spiegate più superficialmente poiché molte righe del codice sono simili o addirittura uguali a quelle dell'ordinamento per ID, verranno spiegate solo le parti in più e le differenze.

Procedura *"sortByPriorita"*:

```
#Sort decrescente per priorita'
sortByPriorita:
ricercaP:
        move $t0, $t8
#Ricerca del massimo fra $t0 e la fine della coda
ricercaMassimoP:
        lw $t1, 24($t0)
        beq $t1, $zero, fineRicercaMassimoP
        lw $t2, 16($t0)
        move $t4,$t0
loopRicercaP:
        beq $t1, $zero, scambiP
                lw $t3, 16($t1)
        bge $t2, $t3,continuaP
        move $t2,$t3
        move $t4,$t1
continuaP:
        lw $t1, 24($t1)
        j loopRicercaP
scambiP:
#Quando abbiamo trovato il valore massimo(puntato da $t4)
#si effettuano gli scambi degli elementi adiacenti per portarlo nella posizione puntata da $t0
        lw $t5, 0($t4)
        bgt $t0, $t5, esciP

scambiaAdiacentiP:
        lw $t6, 16($t4)
        lw $t7, 16($t5)
        beq $t6, $t7, nonScambiareP

        lw  $t6, 4($t5)
        lw $t7, 4($t4)
        sw $t6, 4($t4)
        sw $t7, 4($t5)

        lw  $t6, 8($t5)
        lw $t7, 8($t4)
        sw $t6, 8($t4)
        sw $t7, 8($t5)
        lw  $t6, 12($t5)
        lw $t7, 12($t4)
        sw $t6, 12($t4)
        sw $t7, 12($t5)

        lw  $t6, 16($t5)
        lw $t7, 16($t4)
        sw $t6, 16($t4)
        sw $t7, 16($t5)

        lw  $t6, 20($t5)
        lw $t7, 20($t4)
        sw $t6, 20($t4)
        sw $t7, 20($t5)

nonScambiareP:
        lw $t4, 0($t4)
        lw $t5, 0($t4)
        bgt $t0,$t5, esciP
        j scambiaAdiacentiP

esciP:
        lw $t0, 24($t0)
        j ricercaMassimoP

fineRicercaMassimoP:
        jr $ra
```

La procedura sortByPriorita ha lo stesso funzionamento della procedura di ordinamento per Id, la differenza è che questa procedura opera confrontando la priorità dei task (16($t0), quinto campo del record) e non l'Id.

Anche la procedura di ordinamento per Priorità esegue gli stessi passi della procedura di ordinamento per Id quali, ricerca del massimo valore e scambi fra elementi adiacenti, per ottenere una coda di task ordinata in base alla priorità.

Procedura *"sortByEsecuzioni"*:

```
611 #Sort crescente per Esecuzioni rimanenti
612 sortByEsecuzioni:
613 ricercaE:
614         move $t0, $t8
615 #Ricerca del minimo fra $t0 e la fine della coda
616 ricercaMinimoE:
617         lw $t1, 24($t0)
618         beq $t1, $zero, fineRicercaMinimoE
619         lw $t2, 20($t0)
620         move $t4,$t0
621 loopRicercaE:
622         beq $t1, $zero, scambiE
623         lw $t3, 20($t1)
624         ble $t2, $t3,continuaE
625         move $t2,$t3
626         move $t4,$t1
627 continuaE:
628         lw $t1, 24($t1)
629         j loopRicercaE
630 scambiE:
631 #Quando abbiamo trovato il valore minimo(puntato da $t4)
632 #si effettuano gli scambi degli elementi adiacenti per portarlo nella posizione puntata da $t0
633         lw $t5, 0($t4)
634         bgt $t0, $t5, esciE
635 scambiaAdiacentiE:
636
637         lw $t6, 20($t4)
638         lw $t7, 20($t5)
639         beq $t6, $t7, nonScambiareP
640
641         lw  $t6, 4($t5)
642         lw $t7, 4($t4)
643         sw $t6, 4($t4)
644         sw $t7, 4($t5)
645
646         lw  $t6, 8($t5)
647         lw $t7, 8($t4)
648         sw $t6, 8($t4)
649         sw $t7, 8($t5)
651         lw  $t6, 12($t5)
652         lw $t7, 12($t4)
653         sw $t6, 12($t4)
654         sw $t7, 12($t5)
655
656         lw  $t6, 16($t5)
657         lw $t7, 16($t4)
658         sw $t6, 16($t4)
659         sw $t7, 16($t5)
660
661         lw  $t6, 20($t5)
662         lw $t7, 20($t4)
663         sw $t6, 20($t4)
664         sw $t7, 20($t5)
665
666 nonScambiareE:
667         lw $t4, 0($t4)
668         lw $t5, 0($t4)
669         bgt $t0,$t5, esciE
670         j scambiaAdiacentiE
671
672 esciE:
673         lw $t0, 24($t0)
674         j ricercaMinimoE
675
676 fineRicercaMinimoE:
677         jr $ra
```

La procedura sortByEsecuzioni ha lo stesso funzionamento delle due procedure precedenti, le differenze sono che questa procedura opera confrontando le esecuzioni rimanenti dei task (20($t0), sesto campo del record) invece che l'Id e come vediamo alla riga 624 è effettuato il controllo opposto rispetto agli altri due metodi di ordinamento, infatti ad ogni ciclo è trovato il task con il numero di esecuzioni rimanenti più basso.

Anche la procedura di ordinamento per esecuzioni rimanenti esegue gli stessi passi delle altre procedure di ordinamento quali, ricerca del minimo valore e scambi fra elementi adiacenti, per ottenere una coda di task ordinata in base alle esecuzioni rimanenti.

Procedura *“stampaTabella”*:

```
682 stampaTabella:
683         li $v0, 4
684         la $a0, primaLineaTab
685         syscall
686
687 stampaRighe:
688         addi $sp,$sp,-4
689         sw $ra,0($sp)
690
691         move $t0,$t8
692 loopStampaRighe:
693 #Stampa ID Task
694         li $v0,4
695         la $a0,inizioStringaDati
696         syscall
697
698         li $v0,1
699         lw $a0, 4($t0)
700         syscall
701
702         slti $t4,$a0,10
703
704         beq $t4,1,idAUnaCifra
705         li $v0,4
706         la $a0,fineStringaDati
707         syscall
708 idAUnaCifra:
709         li $v0,4
710         la $a0,fineStringaDatiDoppi
711         syscall
712 #Stampa Proprieta'  del Task
713         li $a0,6
714         jal stampaNSpazi
715
716         li $v0,1
717         lw $a0, 16($t0)
718         syscall
```

```
708 idAUnaCifra:
709         li $v0,4
710         la $a0,fineStringaDatiDoppia
711         syscall
712 #Stampa Proprieta'  del Task
713         li $a0,6
714         jal stampaNSpazi
715
716         li $v0,1
717         lw $a0, 16($t0)
718         syscall
719
720         li $a0,5
721         jal stampaNSpazi
722
723         li $v0,4
724         la $a0,stanga
725         syscall
726
727 # Stampa Nome del Task
728         li $a0,2
729         jal stampaNSpazi
730 # Stampa del nome per caratteri
731         la $s0, 8($t0)
732         li $t6, 0
733 stampaCaratteri:
734         lb $t4,($s0)
735         beq $t4,10, esciStampaCaratteri
736         beq $t4,3, esciStampaCaratteri
737         beq $t4,0, esciStampaCaratteri
738         li $v0, 11
739         move $a0,$t4
740         syscall
741         addi $t6,$t6,1
742         addi $s0,$s0,1
743         j stampaCaratteri
744 esciStampaCaratteri:
```

```
744 esciStampaCaratteri:
745         li $t5,11
746         sub $t5,$t5,$t6
747         move $a0,$t5
748         jal stampaNSpazi
749
750         li $v0,4
751         la $a0,stanga
752         syscall
753 #Stampa L'esecuzioni rimanenti del Task
754         li $a0,10
755         jal stampaNSpazi
756         lw $a0,20($t0)
757         li $v0,1
758         syscall
759         li $t5,11
760         slti $t4,$a0,10
761         add $t5,$t5,$t4
762         move $a0,$t5
763         jal stampaNSpazi
764         li $v0,4
765         la $a0,stangaFineTab
766         syscall
767         li $v0,4
768         la $a0,ultimaLienaTab
769         syscall
770
771         lw $t2, 24($t0)
772         beq $t2, $zero, fineStampaTabelle
773
774         lw $t0, 24($t0)
775         j loopStampaRighe
776
777 fineStampaTabelle:
778         lw $ra,0($sp)
779         addi $sp,$sp,4
780         jr $ra
```

Questa procedura permette la stampa di una tabella con i valori dei task memorizzati nell’heap, inizialmente stampiamo la stringa ‘primaLineaTab’ che è la stringa di inizio della tabella e dopo aver caricato il puntatore alla coda in $t0 e salvato il registro $ra nello stack per richiamare altre funzioni, iniziamo con la stampa dei campi.

Il primo campo da stampare è l’Id del task (4($t0)) stampato tramite una Syscall 1, la gestione del numero degli spazi è effettuata controllando se l’id è a una cifra o due e stampando uno spazio in meno nel secondo caso.

La stampa della priorità (16($t0)) è effettuata anch’essa tramite una Syscall 1, stampando prima del valore 6 spazi e dopo 5 tramite la funzione stampaNSpazi, in questo caso non sono effettuati controlli essendo la priorità sempre di una cifra.

Il nome del task (8($t0)) viene stampato carattere per carattere a partire dalla riga 727, calcolando il numero di caratteri stampati, aumentando il registro $t5 ad ogni carattere stampato e sottraendo questo valore ad 11 troveremo il numero di spazi da stampare e tramite la funzione stampaNSpazi li stamperemo a video.

L’ultimo valore da stampare è il numero di esecuzioni rimanenti (20($t0)) ed anche in questo caso controlleremo se il numero è a due cifre per stampare uno spazio in meno, dopo aver stampato un record e la linea di divisione controlleremo se il puntatore al record successivo è 0, in questo caso usciamo dalla funzione, mentre se è diverso da 0 saltiamo all’etichetta loopStampaRighe per stampare un altro task.

Procedura *"stampaNSpazi"*:

```
786 #Funzione che stampa un numero di spazi pari al valore contenuto in $a0
787 #Parametro $a0 numero di spazi
788 stampaNSpazi:
789         move $t7,$a0
790 stampaSpazi:
791         ble $t7,$zero,fineStampaSpazi
792         li $v0,4
793         la $a0,spazio
794         syscall
795         addi $t7, $t7, -1
796         j stampaSpazi
797 fineStampaSpazi:
798         jr $ra
799
```

Questa procedura è utilizzata dalla procedura stampaTabella e permette di stampare un numero di spazi pari al numero passato come parametro in $a0, è composta da un ciclo che decrementa il numero di spazi e ne stampa uno fino a che raggiungiamo il numero 0 in $t7 ed usciamo dal ciclo tornando al chiamante.

Procedura *"errScelta"*:

```
816 #Errori
817
818 errScelta:
819         li $v0,4
820         la $a0, erroreScelta
821         syscall
822         j sceltaMenu
823
```

L'errore di scelta sbagliata del menu principale è gestito saltando a questa funzione, essa stamperà la stringa che informa l'utente di aver inserito un valore errato e permetterà, attraverso il salto all'etichetta sceltaMenu, di scegliere nuovamente l'operazione da fare nel menu principale.

Evoluzione dell'heap.

In questo capitolo mostreremo l'evoluzione della memoria heap nel caso in cui si inserisca, si rimuova un task o si cambi la politica di scheduling utilizzata.

Inserimento di 3 task:

ID	PRIORITA'	NOME TASK	ESECUZION. RIMANENTI
0	7	Task0	2
1	6	Task1	2
2	5	Task2	1

Inizialmente la politica di scheduling utilizzata è quella su priorità, quindi avremo i valori di priorità ordinati in modo decrescente e il task da eseguire sarà Task2, ovvero quello con priorità minore.

Memorizzazione dei task nell'heap:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268697600	0	0	1802723668	2608	7	2	268697628	268697600
268697632	1	1802723668	2609	6	2	268697656	268697628	2
268697664	1802723668	2610	5	1	0	0	0	0

Cambio della politica di scheduling:

ID	PRIORITA'	NOME TASK	ESECUZION. RIMANENTI
2	5	Task2	1
1	6	Task1	2
0	7	Task0	2

Il cambiamento della politica di scheduling cambierà il modo in cui sono ordinati i task, dopo il cambio verranno ordinati per esecuzioni rimanenti in modo crescente e il task da eseguire sarà Task0, ovvero quello con esecuzioni rimanenti maggiori.

In questo caso abbiamo due task con lo stesso valore di esecuzioni rimanenti, i quali saranno ordinati in ordine decrescente rispetto all'id.

Memorizzazione dei task nell'heap:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268697600	0	2	1802723668	2610	5	1	268697628	268697600
268697632	1	1802723668	2609	6	2	268697656	268697628	0
268697664	1802723668	2608	7	2	0	0	0	0

Rimozione di Task1:

ID	PRIORITA'	NOME TASK	ESECUZION. RIMANENTI
2	5	Task2	1
0	7	Task0	2

La rimozione del Task1 copia il primo record della coda (Task2) sul secondo (Task1) e riordina in base alla politica di scheduling utilizzata.

Memorizzazione dei task nell'heap:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268697600	0	0	0	0	0	0	0	0
268697632	2	1802723668	2610	5	1	268697656	268697628	0
268697664	1802723668	2608	7	2	0	0	0	0