



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

**Architetture  
degli Elaboratori**

## Relazione per il Progetto di Laboratorio

Anno Accademico 2015/2016

??? ????? ??????? ????@stud.unifi.it  
??? ????? ??????? ????@stud.unifi.it  
??? ????? ??????? ????@stud.unifi.it

Data di consegna: ??/??/????

### Esercizio 1: Simulatore di chiamate a procedura

#### Descrizione ad alto livello

Iniziamo la descrizione del primo esercizio con la descrizione ad alto livello del codice. La descrizione sarà proposta sia tramite commenti che attraverso dello pseudocodice. All'interno dello pseudocodice verranno definite tutte le procedure definite nel codice assembly, mentre altre funzioni e comandi primitivi, dal significato intuitivo, verranno indicati in blu.

L'idea generale del programma che realizza la soluzione a questo primo esercizio è di analizzare la stringa e, a seconda dell'operazione aritmetica che la caratterizza, invocare l'opportuna procedura. La procedura di ogni operazione si occuperà di estrarre le sotto-stringhe relative ai suoi due operandi e di ricavarne il valore, invocando su di esse, ricorsivamente, la procedura che analizza una stringa. Una volta ricavati i valori degli operandi, ogni procedura potrà combinarli a seconda dell'operazione che implementa e restituire il risultato finale.

Per gestire lo scorrimento della stringa e delle varie sotto-stringhe si è scelto di mantenere due puntatori, uno che punta al primo carattere della stringa in esame ed uno che punta all'ultimo.

Il punto d'entrata del codice è il `main()`, descritto qui di seguito:

```
main(){
2   file_descriptor = open("chiamate.txt")
   buffer_pointer, length = read( file_descriptor )
4   close( file_descriptor )

6   start = buffer_pointer + 1
   end = buffer_pointer + length - 2
8   depth = 0

10  analyze( start , end, depth)

12  exit()
```

```
}
```

Come si può intuire, il programma apre il file `chiamate.txt` e ne legge il contenuto. Quindi, calcola i puntatori d'inizio e di fine ed invoca la procedura `analyze()` sull'intera stringa. Oltre ai puntatori d'inizio e di fine della stringa, viene mantenuto anche un valore di profondità delle chiamate ricorsive, inizializzato a 0, utile per stampare i messaggi su console con la giusta indentazione.

Di seguito, mostriamo lo pseudocodice della procedura `analyze()` :

```
analyze( start , end, depth){
2   char = load_char(start)

4   switch(char){
      case 's':
6       char2 = load_char(start+2)
        switch(char2){:
8           case 'm':
              res = sum(start, end, depth)
10          default:
              res = sub(start, end, depth)
12        }
      case 'p':
14        res = prod(start, end, depth)
      case 'd':
16        res = div(start, end, depth)
      default:
18        res = 0
        while( start < end){
20            digit = load_char(start) - 48
            res = res + digit
22            res = res * 10
            start = start + 1
24        }
        digit = load_char(start) - 48
26        res = res + digit
28    }

30    return res
}
```

Il compito della funzione `analyze()` è quello di determinare quale operazione aritmetica caratterizza la stringa passata in input (ovvero la stringa delimitata dai puntatori `start` ed `end` passati come parametri). Dal momento che si assume che la stringa descritta in `chiamate.txt` sia sempre sintatticamente corretta, per determinare l'operazione principale della stringa basterà analizzare il primo carattere: se è *s* allora è o una somma o una sottrazione (in questo caso esamina il terzo carattere), se è *p* allora è un prodotto, se è *d* allora è una divisione, altrimenti è un valore già ridotto a intero.

Una volta individuata l'operazione, si passano gli stessi parametri passati ad `analyze()` alla funzione corrispondente all'operazione. Il risultato di questa chiamata a funzione sarà poi restituito a sua volta da `analyze()`.

Nel caso di un valore intero, per poter calcolare il valore di ritorno è necessario effettuare un'operazione di parsing da una serie di caratteri (le cifre che compongono il numero nella stringa) ad un intero. Per fare questo viene innanzitutto caricato ogni carattere, corrispondente ad una

cifra, e viene convertito in intero sottraendovi 48 al suo valore numerico: questo viene fatto in quanto, all'interno della codifica ASCII, le cifre vengono codificate a partire dal valore decimale 48 (corrispondente allo 0) fino a 57 (corrispondente al carattere 9)<sup>1</sup>. A questo punto, se ancora non si è raggiunto l'ultimo carattere, si somma tale valore al numero fin'ora calcolato (inizializzato a 0) e si moltiplica il tutto per 10 (ovvero shiftando, di fatto, di una posizione verso sinistra il valore decimale del numero). Infine, una volta trovata l'ultima cifra, si somma al numero calcolato (senza moltiplicare per 10, essendo le unità) ottenendo il valore finale rappresentato come intero.

Vediamo adesso, di seguito, lo pseudocodice corrispondente alle quattro operazioni aritmetiche:

```
sum(start, end, depth){
2   print_call ( start , end, depth)

4   start = start + 6
   end = end - 1

6   op1, op2 = get_operands(start, end, depth)
8   res = op1 + op2

10  print_return ("somma-return", res, depth)

12  return res
}

14
sub(start, end, depth){
16  print_call ( start , end, depth)

18  start = start + 12
   end = end - 1

20  op1, op2 = get_operands(start, end, depth)
22  res = op1 - op2

24  print_return ("sottrazione-return", res, depth)

26  return res
}

28
prod(start, end, depth){
30  print_call ( start , end, depth)

32  start = start + 9
   end = end - 1

34  op1, op2 = get_operands(start, end, depth)
36  res = op1 * op2

38  print_return ("prodotto-return", res, depth)

40  return res
}

42
div(start, end, depth){
```

---

<sup>1</sup> Si veda la discussione all'indirizzo <http://www.dreamincode.net/forums/topic/284141-how-to-convert-a-char-into-int/>.

```

44     print_call ( start , end, depth)

46     start = start + 10
    end = end - 1

48

50     op1, op2 = get_operands(start, end, depth)
    res = op1 / op2

52     print_return("divisione-return", res, depth)

54     return res
}

```

Le implementazioni delle quattro operazioni sono molto simili e si distinguono soltanto per alcuni dettagli. Innanzitutto viene invocata la procedura `print_call()` sugli stessi parametri di input: questa funzione permette di stampare su console la riga corrispondente alla chiamata a procedura. Dopodiché vengono aggiornati i puntatori d'inizio e di fine della stringa saltando il nome dell'operazione in testa e l'apertura e chiusura delle parentesi: il puntatore finale viene sempre decrementato di 1 (per saltare la chiusura di parentesi finale) mentre i caratteri da saltare all'inizio variano a seconda dell'operazione (ad esempio per la somma si deve saltare *somma*(, ovvero 6 caratteri, come vediamo in riga 4 del codice, mentre per la sottrazione si salta *sottrazione*(, ovvero 12 caratteri totali, come si vede in riga 18). A questo punto si invoca la funzione `get_operands()`, sui nuovi puntatori aggiornati e sulla stessa profondità, che si occupa di calcolare il valore intero dei due operandi coinvolti nell'operazione e restituirli. Una volta ottenuti gli operandi, si può effettuare l'operazione richiesta (somma, sottrazione, ecc...). Quindi verrà invocata la procedura `print_return()` la quale, data una stringa caratterizzante l'operazione, il risultato calcolato e la profondità, stampa il messaggio su console relativo al ritorno della procedura. Infine la funzione restituisce il valore calcolato.

Vediamo di seguito l'implementazione in pseudocodice della funzione `get_operands()` :

```

get_operands(start, end, depth){
2     depth = depth + 1
    i = start
    pars = 0
4     while(true){
6         char = load_char(i)
        switch(char){
8             case '(':
                pars = pars + 1
10            case ')':
                pars = pars - 1
12            case ',':
                if(pars == 0){
14                break
                }
16        }
        i = i + 1
18    }
    res1 = analyze(start , i-1, depth)
20    res2 = analyze(i+1, end, depth)
    return res1, res2
22 }

```

La funzione che estrae il valore degli operandi incrementa, innanzitutto, la profondità di chiamata di 1: infatti, quando si effettueranno le chiamate ricorsive sui due operandi, queste avranno

una profondità maggiore rispetto alla chiamata “padre”. Quindi, viene creato un puntatore `i`, inizializzato col puntatore d’inizio, ed un contatore `pars`, inizializzato a 0, che conta il numero di parentesi aperte ma non chiuse.

Dopodiché si inizia un ciclo. Il ciclo consiste nel caricare il carattere attualmente puntato dal puntatore `i` e controllare, innanzitutto, che non sia una parentesi, nel qual caso incrementa o decrementa `pars` di conseguenza e incrementa `i`. Se il carattere è invece una virgola, controlla allora se `pars` è 0: in questo caso si è trovata la posizione della virgola che separa i due operandi dell’operazione più esterna, in caso contrario, invece, è la virgola di un’operazione più interna, che non ci interessa al momento.

Una volta trovata la virgola che divide gli operandi dell’operazione in esame si esce dal ciclo e si possono invocare le chiamate ricorsive sui due operandi per ottenerne i valori: il primo è delimitato dal puntatore d’inizio e dal puntatore precedente a quello della virgola, mentre il secondo è delimitato dal puntatore successivo alla virgola e dal puntatore di fine. I due valori verranno quindi restituiti, in modo da poter essere combinati opportunamente dalla funzione dell’operazione aritmetica, come visto prima.

Infine, vediamo l’implementazione delle due funzioni ausiliarie `print_call()` e `print_return()`:

```
print_call( start , end, depth){
2   while(depth > 0){
      print_tab()
4     depth = depth - 1
    }
6   print("-->")
   while( start <= end){
8     char = load_char(start)
      print(char)
10    start = start + 1
    }
12   print_newline()
  }

14
print_return( return_string , res , depth){
16   while(depth > 0){
      print_tab()
18     depth = depth - 1
    }
20   print("<--" + return_string + "(" + res + ")")
   print_newline()
22 }
```

Entrambe le procedure iniziano stampando un numero di tabulazioni pari alla profondità di chiamata. Dopodiché viene stampata la freccia, verso destra o verso sinistra, a seconda che sia l’invocazione o la terminazione di una chiamata, rispettivamente. Infine, per l’invocazione di chiamata, viene effettuato un ciclo per stampare l’intera stringa attuale, mentre per la terminazione di chiamata viene stampata la stringa caratteristica dell’operazione (ad esempio `"somma-return"` per la somma) e quindi il valore del risultato tra parentesi.

## Motivazione delle scelte implementative

La principale scelta implementativa, ovvero mantenere di volta in volta i puntatori d’inizio e di fine, è stata una scelta dettata dalla semplicità e dalla facilità d’implementazione, nonché da motivazioni legate alla performance del codice. Una possibile alternativa sarebbe infatti potuta essere quella di andare effettivamente di volta in volta a modificare la stringa, shiftandola verso sinistra per eliminare i caratteri in testa e shiftando a sinistra il carattere di terminazione della

stringa per eliminare i caratteri in coda. Questa soluzione alternativa sarebbe risultata chiaramente più macchinosa e difficile da implementare, nonché più pesante a livello di esecuzione, dovendo ogni volta, anche per eliminare un solo carattere, shiftare l'intera stringa, rendendo ogni operazione sulla stringa un'operazione di complessità  $\mathcal{O}(n)$ , con  $n$  lunghezza della stringa. Lavorando sui puntatori, invece, ogni operazione sulla stringa ha costo lineare  $\mathcal{O}(1)$  e l'implementazione di questa strategia è sicuramente più semplice e leggibile, in quanto uno shift della stringa di qualsiasi tipo corrisponde al semplice incremento/decremento del puntatore corrispondente.

## Uso di registri e memoria

La stringa letta dal file viene allocata, per comodità, nella sezione della memoria statica, assumendo un massimo di 1024 byte, ovvero 1024 caratteri.

Prima di ogni chiamata a procedura, come da convenzione, viene allocato spazio sufficiente nello stack frame, in modo da poter memorizzare e “mettere al sicuro” i valori che si intende recuperare dopo la terminazione della procedura invocata.

In alcuni punti, come ad esempio all'inizio di una procedura, vengono fatte delle copie, da un registro ad un altro, che in alcuni casi possono sembrare anche troppo ridondanti o inutili. Questa scelta è stata fatta in modo da rendere il codice più leggibile ma soprattutto per seguire nel modo più rigoroso possibile le convenzioni sull'uso dei registri: in alcuni casi, ad esempio, si potrebbe lavorare direttamente su registri come `$a0`, `$a1` o `$v0`, ma le convenzioni impongono che tali registri sono riservati agli input e agli output, ed è quindi necessario, nel caso si volesse lavorare con valori contenuti in essi, copiarne preventivamente il contenuto su registri temporanei, come `$t0` e quindi effettuare le operazioni necessarie. Inoltre sono state seguite attentamente le convenzioni che distinguono l'utilizzo tra registri temporanei (`$t0` - `$t9`) e registri da preservare (`$s0` - `$s7`).

L'evoluzione tipica dello stack parte con una chiamata ad `analyze()`, alla quale si sussegue una chiamata alla funzione che implementa l'operazione riscontrata, come `sum()` oppure `prod()`. Dopodiché a queste segue una chiamata a `get_operands()`, la quale genera, in sequenza, due nuove chiamate ad `analyze()`. Le chiamate seguono questo pattern finché `analyze()` non incappa in un valore intero: in questo caso inizia il processo di backtracking fino alla prima chiamata `get_operands()`, la quale potrà invocare un secondo `analyze()` (se quello era il primo) o proseguire col backtracking a sua volta (se era la seconda chiamata).

## Simulazioni

Mostriamo adesso un paio di esecuzioni tipo del programma. Dal momento che si assume che le stringhe di input siano sempre sintatticamente corrette, non verranno presi in esame situazioni di errore in cui le stringhe inserite hanno sintassi scorrette. Verranno invece mostrati gli output per le tre stringhe proposte nel testo dell'esercizio.

Come possiamo notare dalle Figure dalla 1 alla 3, i risultati stampati su console sono esattamente quelli attesi, indice che il programma funziona correttamente per input dalle dimensioni più svariate. Inoltre si può notare come l'indentazione delle varie chiamate sia stata implementata con successo, rendendo l'output più chiaro ed intuitivo.

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

Figura 1: Output con la stringa *"divisione(5,2)"*.

```
Console
-->prodotto(prodotto(3,4),2)
-->prodotto(3,4)
<--prodotto-return(12)
<--prodotto-return(24)
```

Figura 2: Output con la stringa *"prodotto(prodotto(3,4),2)"*.

```
Console
-->somma(7,somma(sottrazione(0,5),prodotto(divisione(7,2),3)))
-->somma(sottrazione(0,5),prodotto(divisione(7,2),3))
-->sottrazione(0,5)
<--sottrazione-return(-5)
-->prodotto(divisione(7,2),3)
-->divisione(7,2)
<--divisione-return(3)
<--prodotto-return(9)
<--somma-return(4)
<--somma-return(11)
```

Figura 3: Output con la stringa *"somma(7,somma(sottrazione(0,5),prodotto(divisione(7,2),3)))"*.

## Codice MIPS

Di seguito, il codice MIPS completo che implementa il programma descritto dall'esercizio 1, opportunamente commentato.

```
1 # Title: Simulatore di chiamate a procedura      Filename: es1.s
2 # Author1: ??? ??????      ???????      ??? ??????@stud.unifi.it
3 # Author2: ??? ??????      ???????      ??? ??????@stud.unifi.it
4 # Author3: ??? ??????      ???????      ??? ??????@stud.unifi.it
5 # Date: ??/??/????
6 # Description: Chiamate a procedura per l'esecuzione di operazioni aritmetiche
```

```

7  # Input: chiamate.txt
8  # Output: Traccia delle chiamate su console
9
10 ##### Data segment #####
11 .data
12 file :      .asciiz "chiamate.txt"
13 tab:        .asciiz "\t"
14 newline:    .asciiz "\n"
15 arrow_r:    .asciiz "-->"
16 arrow_l:    .asciiz "<--"
17 buffer:     .space 1024
18 sum_return: .asciiz "somma-return"
19 sub_return: .asciiz "sottrazione-return"
20 prod_return: .asciiz "prodotto-return"
21 div_return: .asciiz "divisione-return"
22
23 ##### Code segment #####
24 .text
25 .globl main
26
27 ### Print call ###
28 print_call :
29     move $t0, $a0 # copia i tre parametri in registri temporanei
30     move $t1, $a1
31     move $t2, $a2
32
33 print_call_tabs:
34     beq $t2, $zero, print_call_arrow # se la profondità è 0, allora procede a stampare
    la stringa
35                                     # altrimenti
36     li $v0, 4 # stampa una tabulazione
37     la $a0, tab
38     syscall
39
40     addi $t2, $t2, -1 # decrementa la profondità
41
42     j print_call_tabs # ed esegue un altro ciclo
43
44 print_call_arrow:
45     li $v0, 4 # stampa la freccia verso destra
46     la $a0, arrow_r
47     syscall
48
49 print_call_string :
50     li $v0, 11 # stampa il carattere puntato da $a0
51     lb $a0, 0($t0)
52     syscall
53
54     beq $t0, $t1, print_call_done # se i puntatori d'inizio e di fine coincidono allora
    la stringa è finita
55
56     addi $t0, $t0, 1 # altrimenti incrementa il puntatore d'inizio (scorre al
    prossimo carattere)
57
58     j print_call_string # ed esegue un altro ciclo
59
60 print_call_done:
61     li $v0, 4 # alla fine stampa una newline (a capo)

```



```

62     la $a0, newline
63     syscall
64
65     jr $ra # e torna al chiamante
66     ### Print call end ###
67
68     ### Print return ###
69     print_return:
70         move $t0, $a0 # copia i tre parametri in registri temporanei
71         move $t1, $a1
72         move $t2, $a2
73
74     print_return_tabs: # analogo a print_call_tabs
75         beq $t2, $zero, print_return_string
76
77         li $v0, 4
78         la $a0, tab
79         syscall
80
81         addi $t2, $t2, -1
82
83         j print_return_tabs
84
85     print_return_string:
86         li $v0, 4 # stampa la freccia verso sinistra
87         la $a0, arrow_l
88         syscall
89         move $a0, $t0 # stampa la stringa di ritorno relativa all'operazione
90         syscall
91         li $v0, 11 # stampa il simbolo di aperta parentesi
92         li $a0, '('
93         syscall
94         li $v0, 1 # stampa il risultato dell'operazione
95         move $a0, $t1
96         syscall
97         li $v0, 11 # stampa il simbolo di chiusa parentesi
98         li $a0, ')'
99         syscall
100        li $v0, 4 # stampa una newline (a capo)
101        la $a0, newline
102        syscall
103
104        jr $ra # infine torna al chiamante
105        ### Print return end ###
106
107        ### Get operands ###
108        get_operands:
109            addi $sp, $sp, -20 # alloca spazio per 5 words nello stack frame
110            sw $ra, 16($sp) # salva l'indirizzo di ritorno
111            sw $a1, 12($sp) # il puntatore di fine
112            addi $a2, $a2, 1 # e la profondità della chiamata incrementata di 1
113            sw $a2, 8($sp)
114
115            move $s0, $a0 # inizializza $s0 (puntatore di scorrimento) con il puntatore d'
            inizio
116            move $t0, $zero # e $t0 con 0 ($t0 indica il numero di parentesi aperte ma non
            ancora chiuse)
117

```

```

118 get_operands_search:
119     lb $t1, 0($s0)           # carica il carattere puntato da $s0
120     li $t2, '('
121     beq $t1, $t2, get_operands_open # controlla se è una parentesi aperta
122     li $t2, ')'
123     beq $t1, $t2, get_operands_close # una parentesi chiusa
124     li $t2, ','
125     beq $t1, $t2, get_operands_comma # oppure una virgola
126     j get_operands_advance        # altrimenti va avanti senza fare niente
127
128 get_operands_open:
129     addi $t0, $t0, 1          # se si trova una parentesi aperta si incrementa $t0
130     j get_operands_advance # e si passa al prossimo carattere
131
132 get_operands_close:
133     addi $t0, $t0, -1         # se si trova una parentesi chiusa si decrementa $t0
134     j get_operands_advance # e si passa al prossimo carattere
135
136 get_operands_comma:
137     beq $t0, $zero, get_operands_found # se si trova una virgola e le parentesi sono
138     bilanciate, allora si è trovato il punto di divisione
139     j get_operands_advance        # altrimenti si passa al prossimo carattere
140
141 get_operands_advance:
142     addi $s0, $s0, 1          # incrementa di 1 il puntatore $s0
143     j get_operands_search      # e continua la ricerca
144
145 get_operands_found:
146     sw $s0, 4($sp) # salva il puntatore alla virgola nello stack
147
148     addi $s0, $s0, -1 # decrementa il puntatore $s0 (carattere subito prima della
149     virgola)
150     move $a1, $s0      # e lo imposta come secondo parametro per analyze
151
152     jal analyze # invoca analyze sul primo operando con profondità incrementata di 1
153
154     sw $v0, 0($sp) # salva il valore del primo operando nello stack
155
156     lw $a0, 4($sp) # recupera il puntatore alla virgola
157     addi $a0, $a0, 1 # e lo imposta come primo parametro (puntatore d'inizio)
158     incrementandolo di 1 (primo carattere dopo la virgola)
159     lw $a1, 12($sp) # recupera il puntatore di fine
160     lw $a2, 8($sp)  # recupera la profondità incrementata di 1
161
162     jal analyze # invoca analyze sul primo operando con profondità incrementata di 1
163
164     move $v1, $v0 # salva il valore del secondo operando come secondo risultato
165     lw $v0, 0($sp) # recupera il valore del primo operando e lo imposta come primo
166     risultato
167
168     lw $ra, 16($sp) # recupera l'indirizzo di ritorno
169     addi $sp, $sp, 20 # dealloca lo stack frame
170     jr $ra           # ritorna al chiamante
171
172 ### Get operands end ###
173
174 ### Call sum ###
175 call_sum:
176     addi $sp, $sp, -20 # alloca spazio per cinque words nello stack frame

```

```

172  sw $ra, 16($sp)    # salva l'indirizzo di ritorno nello stack
173
174  move $s0, $a0 # primo parametro: puntatore d'inizio
175  move $s1, $a1 # secondo: puntatore di fine
176  move $s2, $a2 # terzo: profondità della chiamata
177
178  sw $s0, 12($sp)    # il puntatore d'inizio
179  sw $s1, 8($sp)     # il puntatore di fine
180  sw $s2, 4($sp)     # e la profondità della chiamata
181
182  move $a0, $s0 # primo parametro: puntatore d'inizio
183  move $a1, $s1 # secondo: puntatore di fine
184  move $a2, $s2 # terzo: profondità della chiamata
185
186  jal print_call     # invoca la procedura per la stampa dell'invocazione (con gli
stessi parametri)
187
188  lw $t0, 12($sp)    # recupera il puntatore d'inizio dallo stack
189  addi $t0, $t0, 6    # salta la stringa "somma(" (6 caratteri)
190  lw $t1, 8($sp)     # recupera il puntatore di fine
191  addi $t1, $t1, -1   # scarta l'ultima parentesi chiusa
192  lw $t2, 4($sp)     # recupera la profondità della chiamata
193
194  move $a0, $t0 # primo parametro: puntatore d'inizio
195  move $a1, $t1 # secondo: puntatore di fine
196  move $a2, $t2 # terzo: profondità della chiamata
197
198  jal get_operands   # invoca la procedura per ottenere gli operandi
199
200  move $t0, $v0 # primo valore di ritorno: valore del primo operando
201  move $t1, $v1 # secondo: valore del secondo operando
202
203  add $s0, $t0, $t1  # somma i due operandi ottenuti
204  sw $s0, 0($sp)     # salva il risultato nello stack
205
206  la $a0, sum_return # carica l'indirizzo della stringa sum_return (primo parametro)
207  move $a1, $s0      # imposta il risultato dell'operazione come secondo parametro
208  lw $a2, 4($sp)     # recupera la profondità della chiamata (terzo parametro)
209
210  jal print_return    # invoca la stampa del risultato con questi tre parametri
211
212  lw $v0, 0($sp)     # recupera il risultato dell'operazione
213  lw $ra, 16($sp)    # recupera l'indirizzo di ritorno
214  addi $sp, $sp, 20   # dealloca lo stack frame
215  jr $ra             # torna al chiamante
216  ### Call sum end ###
217
218  ### Call subtraction ###
219  call_sub:
220  addi $sp, $sp, -20 # alloca spazio per cinque words nello stack frame
221  sw $ra, 16($sp)    # salva l'indirizzo di ritorno nello stack
222
223  move $s0, $a0 # primo parametro: puntatore d'inizio
224  move $s1, $a1 # secondo: puntatore di fine
225  move $s2, $a2 # terzo: profondità della chiamata
226
227  sw $s0, 12($sp)    # il puntatore d'inizio
228  sw $s1, 8($sp)     # il puntatore di fine

```

```

229     sw $s2, 4($sp)      # e la profondità della chiamata
230
231     move $a0, $s0 # primo parametro: puntatore d'inizio
232     move $a1, $s1 # secondo: puntatore di fine
233     move $a2, $s2 # terzo: profondità della chiamata
234
235     jal print_call      # invoca la procedura per la stampa dell'invocazione (con gli
stessi parametri)
236
237     lw $t0, 12($sp)     # recupera il puntatore d'inizio dallo stack
238     addi $t0, $t0, 12   # salta la stringa "sottrazione(" (12 caratteri)
239     lw $t1, 8($sp)     # recupera il puntatore di fine
240     addi $t1, $t1, -1   # scarta l'ultima parentesi chiusa
241     lw $t2, 4($sp)     # recupera la profondità della chiamata
242
243     move $a0, $t0 # primo parametro: puntatore d'inizio
244     move $a1, $t1 # secondo: puntatore di fine
245     move $a2, $t2 # terzo: profondità della chiamata
246
247     jal get_operands   # invoca la procedura per ottenere gli operandi
248
249     move $t0, $v0 # primo valore di ritorno: valore del primo operando
250     move $t1, $v1 # secondo: valore del secondo operando
251
252     sub $s0, $t0, $t1  # sottrae il secondo operando al primo
253     sw $s0, 0($sp)     # salva il risultato nello stack
254
255     la $a0, sub_return # carica l'indirizzo della stringa sub_return (primo parametro)
256     move $a1, $s0      # imposta il risultato dell'operazione come secondo parametro
257     lw $a2, 4($sp)     # recupera la profondità della chiamata (terzo parametro)
258
259     jal print_return   # invoca la stampa del risultato con questi tre parametri
260
261     lw $v0, 0($sp)     # recupera il risultato dell'operazione
262     lw $ra, 16($sp)    # recupera l'indirizzo di ritorno
263     addi $sp, $sp, 20  # dealloca lo stack frame
264     jr $ra            # torna al chiamante
265 ### Call subtraction end ###
266
267 ### Call product ###
268 call_prod:
269     addi $sp, $sp, -20 # alloca spazio per cinque words nello stack frame
270     sw $ra, 16($sp)    # salva l'indirizzo di ritorno nello stack
271
272     move $s0, $a0 # primo parametro: puntatore d'inizio
273     move $s1, $a1 # secondo: puntatore di fine
274     move $s2, $a2 # terzo: profondità della chiamata
275
276     sw $s0, 12($sp)    # il puntatore d'inizio
277     sw $s1, 8($sp)     # il puntatore di fine
278     sw $s2, 4($sp)     # e la profondità della chiamata
279
280     move $a0, $s0 # primo parametro: puntatore d'inizio
281     move $a1, $s1 # secondo: puntatore di fine
282     move $a2, $s2 # terzo: profondità della chiamata
283
284     jal print_call      # invoca la procedura per la stampa dell'invocazione (con gli
stessi parametri)

```

```

285
286 lw $t0, 12($sp)    # recupera il puntatore d'inizio dallo stack
287 addi $t0, $t0, 9    # salta la stringa "prodotto(" (9 caratteri)
288 lw $t1, 8($sp)     # recupera il puntatore di fine
289 addi $t1, $t1, -1   # scarta l'ultima parentesi chiusa
290 lw $t2, 4($sp)      # recupera la profondità della chiamata
291
292 move $a0, $t0 # primo parametro: puntatore d'inizio
293 move $a1, $t1 # secondo: puntatore di fine
294 move $a2, $t2 # terzo: profondità della chiamata
295
296 jal get_operands # invoca la procedura per ottenere gli operandi
297
298 move $t0, $v0 # primo valore di ritorno: valore del primo operando
299 move $t1, $v1 # secondo: valore del secondo operando
300
301 mul $s0, $t0, $t1 # moltiplica i due operandi ottenuti
302 sw $s0, 0($sp)    # salva il risultato nello stack
303
304 la $a0, prod_return # carica l'indirizzo della stringa prod_return (primo parametro)
305 move $a1, $s0       # imposta il risultato dell'operazione come secondo parametro
306 lw $a2, 4($sp)      # recupera la profondità della chiamata (terzo parametro)
307
308 jal print_return    # invoca la stampa del risultato con questi tre parametri
309
310 lw $v0, 0($sp)      # recupera il risultato dell'operazione
311 lw $ra, 16($sp)     # recupera l'indirizzo di ritorno
312 addi $sp, $sp, 20   # dealloca lo stack frame
313 jr $ra              # torna al chiamante
314 ### Call product end ###
315
316 ### Call division ###
317 call_div:
318 addi $sp, $sp, -20 # alloca spazio per cinque words nello stack frame
319 sw $ra, 16($sp)    # salva l'indirizzo di ritorno nello stack
320
321 move $s0, $a0 # primo parametro: puntatore d'inizio
322 move $s1, $a1 # secondo: puntatore di fine
323 move $s2, $a2 # terzo: profondità della chiamata
324
325 sw $s0, 12($sp)    # il puntatore d'inizio
326 sw $s1, 8($sp)     # il puntatore di fine
327 sw $s2, 4($sp)     # e la profondità della chiamata
328
329 move $a0, $s0 # primo parametro: puntatore d'inizio
330 move $a1, $s1 # secondo: puntatore di fine
331 move $a2, $s2 # terzo: profondità della chiamata
332
333 jal print_call      # invoca la procedura per la stampa dell'invocazione (con gli
stessi parametri)
334
335 lw $t0, 12($sp)    # recupera il puntatore d'inizio dallo stack
336 addi $t0, $t0, 10  # salta la stringa "divisione(" (10 caratteri)
337 lw $t1, 8($sp)     # recupera il puntatore di fine
338 addi $t1, $t1, -1  # scarta l'ultima parentesi chiusa
339 lw $t2, 4($sp)     # recupera la profondità della chiamata
340
341 move $a0, $t0 # primo parametro: puntatore d'inizio

```

```

342  move $a1, $t1 # secondo: puntatore di fine
343  move $a2, $t2 # terzo: profondità della chiamata
344
345  jal get_operands # invoca la procedura per ottenere gli operandi
346
347  move $t0, $v0 # primo valore di ritorno: valore del primo operando
348  move $t1, $v1 # secondo: valore del secondo operando
349
350  div $s0, $t0, $t1 # divide (operazione quoziente) il primo operando per il secondo
351  sw $s0, 0($sp)    # salva il risultato nello stack
352
353  la $a0, div_return # carica l'indirizzo della stringa div_return (primo parametro)
354  move $a1, $s0      # imposta il risultato dell'operazione come secondo parametro
355  lw $a2, 4($sp)     # recupera la profondità della chiamata (terzo parametro)
356
357  jal print_return   # invoca la stampa del risultato con questi tre parametri
358
359  lw $v0, 0($sp)     # recupera il risultato dell'operazione
360  lw $ra, 16($sp)    # recupera l'indirizzo di ritorno
361  addi $sp, $sp, 20  # dealloca lo stack frame
362  jr $ra             # torna al chiamante
363  ### Call division end ###
364
365  ### Analyze ###
366  analyze:
367      addi $sp, $sp, -4 # alloca spazio per una word nello stack frame
368      sw $ra, 0($sp)   # salva l'indirizzo di ritorno del chiamante
369
370      # i parametri di input delle chiamate sono gli stessi di analyze, ovvero:
371      # primo: puntatore d'inizio
372      # secondo: puntatore di fine
373      # terzo: profondità della chiamata
374
375      lb $t0, 0($a0)   # carica il primo carattere della stringa
376      li $t1, 's'      # se è 's'
377      beq $t0, $t1, analyze_sum_sub # allora è una somma o una sottrazione
378      li $t1, 'p'      # se è 'p'
379      beq $t0, $t1, jump_prod # allora è un prodotto
380      li $t1, 'd'      # se è 'd'
381      beq $t0, $t1, jump_div # allora è una divisione
382      j analyze_int    # altrimenti è già un intero
383
384  analyze_sum_sub:
385      lb $t0, 2($a0)   # carica il terzo carattere della stringa
386      li $t1, 'm'      # se è 'm'
387      beq $t0, $t1, jump_sum # allora è una somma
388      j jump_sub        # altrimenti è una sottrazione
389
390  jump_sum:
391      jal call_sum     # invoca la procedura relativa alla somma
392      j analyze_end    # quando ritorna invoca la fine dell'analisi (comune a tutte le
                        # operazioni)
393
394  jump_sub:           # sub, prod e div analoghe a sum
395      jal call_sub
396      j analyze_end
397
398  jump_prod:

```

```

399     jal call_prod
400     j analyze_end
401
402 jump_div:
403     jal call_div
404     j analyze_end
405
406 analyze_int:
407     move $v0, $zero #inizializza il valore dell'intero a 0
408
409 analyze_int_loop:
410     lb $t1, 0($a0)    # carica il carattere puntato da $a0
411     addi $t1, $t1, -48 # sottrae 48 (parsing da codifica ASCII a intero)
412     add $v0, $v0, $t1 # somma la cifra ottenuta ($t1) col valore fin'ora calcolato (
413     $v0)
414
415     beq $a0, $a1, analyze_end # se i puntatori d'inizio e fine coincidono allora era l'
416     ultimo carattere
417
418     # altrimenti
419     li $t2, 10
420     mul $v0, $v0, $t2 # moltiplica il valore fin'ora calcolato per 10
421     addi $a0, $a0, 1 # incrementa il puntatore di 1 (prossimo carattere)
422
423     j analyze_int_loop # ed esegue un altro ciclo
424
425 analyze_end:
426     # nel caso delle chiamate a procedura, il valore di ritorno è lo stesso (valore
427     dell'espressione)
428     lw $ra, 0($sp)    # recupera l'indirizzo di ritorno
429     addi $sp, $sp, 4 # dealloca lo stack
430     jr $ra            # torna al chiamante
431
432 ### Analyze end ###
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

le " finali)
454     li $t4, 0           # e la profondità delle chiamate iniziale
455
456     move $a0, $t2 # prepara i parametri
457     move $a1, $t3
458     move $a2, $t4
459
460     jal analyze # inizia analizzando l'intera stringa
461                # ignora il valore di ritorno
462
463     li $v0, 10 # uscita dal programma (syscall)
464     syscall
465     ### Main end ###

```

## Esercizio 2: Scheduler di processi

### Descrizione ad alto livello

Come per l'esercizio precedente, descriviamo il funzionamento generale del programma tramite una descrizione ad alto livello.

Essendo questo esercizio, rispetto al precedente, molto più complesso, la descrizione ad alto livello del codice trascurerà alcuni dettagli meno interessanti, come la stampa su console di messaggi di servizio, e si concentrerà invece su aspetti più importanti del codice, per rendere la lettura più chiara e scorrevole.

L'idea generale è quella di costruire e gestire due liste doppiamente concatenate, una corrispondente alla politica di scheduling A (su priorità) ed una corrispondente alla politica B (su esecuzioni rimanenti). Ad ogni task corrisponderà quindi un record, opportunamente allocato in memoria in modo dinamico, composto da una serie di "campi". Oltre ai campi caratteristici del task, come ID, nome, ecc., questo record conterrà anche quattro puntatori ad altri task, ovvero i puntatori al precedente e successivo sia per la politica A che per la B. Servirà quindi mantenere, in ogni momento, due puntatori, che rappresentano i puntatori d'inizio delle due liste concatenate, e la politica di scheduling attuale. Per semplicità di esposizione, nello pseudocodice rappresenteremo l'accesso ai campi di ogni record con una sintassi simile all'accesso ai campi di un oggetto (ovvero l'istanza di una classe): questa è solamente una scorciatoia a livello di esposizione per semplificare la lettura dello pseudocodice e non vuol dare l'idea che si stanno implementando classi ed oggetti. Le code, inoltre, saranno implementate al contrario, ovvero gli elementi puntati dai puntatori d'inizio sono di fatto gli elementi in fondo alle due code. Il perché di questa scelta verrà commentato in seguito.

Detto questo, iniziamo col vedere, qui di seguito, la descrizione ad alto livello del segmento di codice `main()`, punto d'entrata del programma:

```

main(){
2     sched = 'a'
    start_A = null
4     start_B = null

6     while(true){
        com = read_int("Inserire un comando: ")
8         switch(com){
            case 1:
10             start_A, start_B = insert_task(start_A, start_B)
            case 2:
12             start_A, start_B = run_first(sched, start_A, start_B)
            case 3:

```



```

14         start_A, start_B = run_id(start_A, start_B)
15     case 4:
16         start_A, start_B = delete_id(start_A, start_B)
17     case 5:
18         start_A, start_B = change_prio(start_A, start_B)
19     case 6:
20         sched = change_sched(sched)
21     case 7:
22         print("Terminazione del programma.")
23         exit()
24     default:
25         print("Menu: ...")
26 }
27
28 if(sched == 'a'){
29     i = start_A
30 } else {
31     i = start_B
32 }
33 delim = "+---+-----+-----+-----+"
34 header = "| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |"
35 println(delim)
36 println(header)
37 println(delim)
38 if(i == null){
39     println("|                Coda vuota!                |")
40     println(delim)
41     continue
42 }
43 while(i != null){
44     print("/ ")
45     id = i.id
46     if (id <= 9){
47         print(" " + id)
48     } else {
49         print(" " + id)
50     }
51     print(" / " + i.prio + " / " + i.name + " / ")
52     cycles = i.cycles
53     if(cycles <= 9){
54         print(" " + cycles)
55     } else {
56         print(cycles)
57     }
58     println(" /")
59     println(delim)
60     if(sched == 'a'){
61         i = i.next_A
62     } else {
63         i = i.next_B
64     }
65 }
66 }

```

La prima cosa che viene fatta è inizializzare i due puntatori iniziali a `null` e la politica di scheduling ad A. Quindi si inizia un ciclo nel quale ad ogni passo viene chiesto un comando

tramite l'inserimento di un intero da input: a questo punto, a seconda dell'intero inserito, viene eseguita l'operazione corrispondente, passandogli i parametri necessari e catturando i valori di ritorno. Ad esempio, la funzione corrispondente al comando 1, ovvero l'inserimento di un nuovo task, prende come parametri i puntatori d'inizio attuali e restituisce una coppia di puntatori d'inizio aggiornati. Se il comando è il 7, allora si procede all'uscita dal programma senza invocare procedure, mentre in tutti gli altri casi (ad esempio inserendo altri interi, caratteri, stringhe o semplicemente premendo Invio) viene stampato il menu dove vengono descritti i vari comandi disponibili.

Una volta eseguita la funzione richiesta, vengono eseguite le istruzioni per la stampa della coda (dalla riga 28 dello pseudocodice). Dapprima, viene inizializzato un puntatore, che servirà a scorrere la coda a seconda della politica di scheduling attuale, e verrà stampata l'intestazione della tabella. In caso di coda vuota, verrà stampato un messaggio all'interno della tabella e si tornerà alla richiesta di selezione di un comando. In caso contrario inizierà invece un ciclo su tutti gli elementi della lista, da quello in fondo a quello in testa. Per ogni elemento viene stampata una riga nella tabella, contenente tutte le informazioni del task, estraendole dal record. Tutti gli spazi aggiuntivi che si vedono nelle varie stringhe che vengono stampate servono a rendere i campi della tabella allineati. Inoltre, su due campi, l'ID e le esecuzioni rimanenti, viene controllato se il numero è a due cifre oppure una sola, in modo da stampare un numero corretto di spazi per allineare il valore a destra (in questo caso, mentre per le esecuzioni rimanenti siamo sicuri di avere al massimo un numero di due cifre, per l'ID non possiamo esserne certi, quindi assumiamo che l'ID di un task arrivi fino a 99, altrimenti la tabella verrà non allineata). Al termine del ciclo, verrà quindi caricato il task seguente nella lista, a seconda della politica di scheduling.

Vediamo adesso, una per una, le implementazioni ad alto livello dei vari comandi, iniziando con l'inserimento di un nuovo task.

```

insert_task(start_A, start_B){
2   task = new_task() // alloca spazio sufficiente con sbrk
   task.prev_A = null
4   task.prev_B = null
   while(true){
6       id = read_int("Inserire l'ID: ")
       if(start_A == null){
8           task.id = id
           break
10      }
       duplicate_task = find_id(id, start_A)
12      if(duplicate_task == null){
           task.id = id
14          break
       }
16      println("Task con ID " + id + "già presente.")
   }
18   name = read_string("Inserire il nome: ")
   task.name = name // inserisce soltanto i primo 8 caratteri della stringa inserita
20   while(true){
       prio = read_int("Inserire la priorità: ")
22       if(prio >= 0 AND prio <= 9){
           task.prio = prio
24           break
       }
26   }
   while(true){
28       cycles = read_int("Inserire i cicli di esecuzione: ")
       if(cycles >= 1 AND cycles <= 99){
30           task.cycles = cycles
       }
   }
}

```

```

32         break
33     }
34     task.next_A = null
35     task.next_B = null
36
37     start_A, start_B = insert(task, start_A, start_B)
38
39     return start_A, start_B
40 }

```

La funzione `insert_task()` riceve come argomenti i due puntatori d'inizio, mentre la politica di scheduling attuale non serve in quanto il task dovrà essere inserito in entrambe le liste indifferentemente dalla politica attuale. Per prima cosa viene allocato spazio per il nuovo task (la pseudoistruzione `new_task()` corrisponde di fatto ad una chiamata `SBRK` di 36 byte). Quindi si procede a chiedere da input i valori dei vari campi per poi inserirli all'interno del record del nuovo task. Se la coda è vuota, l'ID viene inserito automaticamente, altrimenti si controlla che non sia già presente un task con lo stesso ID (tramite la funzione `find_id()`, di cui vedremo a breve l'implementazione): se il risultato è positivo, allora si chiede un nuovo ID, altrimenti si può salvare l'ID selezionato all'interno del record. Per quanto riguarda priorità e cicli d'esecuzione viene fatto un controllo simile: si richiede infatti l'inserimento di una priorità tra 0 e 9 e di cicli di esecuzioni tra 1 e 99. I puntatori a precedenti e successivi, invece, vengono inizializzati a `null`, in quanto il task, di fatto, non è ancora inserito nelle liste.

Per inserire il task nelle due code si invoca quindi la funzione `insert()` (che descriveremo a breve) passandogli come parametri il puntatore al task appena creato e i due puntatori d'inizio. Il risultato sarà una coppia di puntatori d'inizio aggiornati, che la funzione `insert_task()` potrà quindi restituire, terminando la sua esecuzione.

Vediamo, di seguito, l'implementazione della funzione ausiliaria `find_id()`:

```

find_id(id, start_A){
2     task = null
3     i = start_A
4     while(i != null){
5         if(id == i.id){
6             task = i
7             break
8         }
9         i = i.next_A
10    }
11    return task
12 }

```

La funzione `find_id()` prende in input un ID ed il puntatore d'inizio della lista A e restituisce il puntatore al task con ID indicato se presente nelle code, `null` altrimenti. Dal momento che, per come abbiamo implementato `insert_task()`, ogni ID è unico nelle code e che gli elementi nelle due code sono esattamente gli stessi (a meno dell'ordinamento), scorrere la lista A o la B è indifferente ai fini di trovare un task con ID specificato. La funzione è molto semplice: si inizializza il puntatore al task da trovare a `null` e quindi si scorre la lista tramite un ciclo dal quale si può uscire soltanto una volta trovato un task con ID corrispondente o una volta terminata la lista. Se il task è stato effettivamente trovato, allora verrà restituito il suo puntatore, altrimenti `null`. Questa funzione è stata implementata a parte in quanto verrà riutilizzata in altri punti del codice.

Parliamo anche l'implementazione della seconda funzione ausiliaria vista fin'ora, ovvero `insert ()` .

La funzione `insert ()` è una funzione molto complessa ed in qualche modo rappresenta il cuore di tutto il programma. L'implementazione di questa funzione è caratterizzata da alcuni costrutti molto particolari, possibili soltanto grazie alle istruzioni di salto messe a disposizione dal linguaggio assembly, la cui conversione in pseudocodice di alto livello è molto complessa, se non impossibile, da rendere pur mantenendo un certo livello di chiarezza e leggibilità. Per queste ragioni commenteremo l'implementazione di questa funzione soltanto a livello testuale e senza l'ausilio di pseudocodice, che renderebbe soltanto le cose più difficili da comprendere.

La funzione `insert ()` è una funzione ausiliare che prende in input il puntatore al task che si vuole inserire nelle due liste ed i puntatori d'inizio delle liste A e B. È quindi una funzione generica per l'inserimento di un task nelle due liste che verrà utilizzata più volte all'interno del programma.

La funzione controlla innanzitutto se le due liste sono vuote (puntatori d'inizio a `null`): in questo caso fa puntare i due puntatori d'inizio al task da inserire e li restituisce. In caso contrario, inizia un primo ciclo, dedicato all'inserimento nella lista A, ovvero alla coda relativa allo scheduling su priorità. Il ciclo procede scorrendo gli elementi della lista A fin tanto che vengono trovati task con priorità maggiore del task da inserire. Se, così facendo, si raggiunge la fine della lista, allora il task viene inserito come ultimo. Se durante lo scorrimento si trova invece un task con priorità uguale al task da inserire, inizia allora un secondo ciclo, più interno, che scorre gli elementi confrontando le loro esecuzioni rimanenti: finché si trovano task con meno esecuzioni (e stessa priorità) del task da inserire, si va avanti, altrimenti si inserisce il task (ovvero non appena si trova una priorità minore o delle esecuzioni rimanenti maggiori o uguali). Se invece, durante il ciclo principale, si raggiunge direttamente un task con priorità minore, allora il task è da inserire tra quel task trovato ed il suo precedente (o come primo della lista se quello era il primo). L'inserimento effettivo del task nella lista A prevede quindi, una volta individuato il punto preciso nella lista in cui inserirlo, di aggiornare i puntatori `prev_A` e `next_A` del task da inserire e del task precedente e successivo al punto in cui si vuole inserire (in più, se il task è da inserire come primo, allora si aggiorna anche il puntatore d'inizio A).

Una volta inserito correttamente il task all'interno della lista A, viene fatto un ciclo del tutto analogo, ed in un certo senso speculare, per l'inserimento nella lista B. Le uniche differenze sono l'utilizzo dei vari puntatori al precedente/successivo, che quindi saranno `prev_B` e `next_B`, ed i criteri di scorrimento della lista: se prima si cercava il primo task con priorità minore di quello da inserire, adesso si cerca invece il primo task con numero di esecuzioni rimanenti strettamente maggiore del task da inserire. Analogamente, se viene trovato un task con le stesse esecuzioni rimanenti, allora inizia un ciclo più interno, in cui si cerca il primo task con priorità minore o uguale (o, sempre, con esecuzioni rimanenti maggiori).

Al termine di questo secondo ciclo, il task risulterà correttamente inserito nelle due liste, trovandosi, per entrambe, esattamente nel suo punto finale, ovvero ordinato secondo la politica di scheduling su priorità (nella lista A) o su esecuzioni rimanenti (lista B). La funzione `insert ()` terminerà quindi restituendo i puntatori d'inizio delle due liste, eventualmente aggiornati.

Passiamo all'implementazione del secondo comando, ovvero l'esecuzione del task in testa alla coda.

```
run_first(sched, start_A, start_B){
2   if(sched == 'a'){
        i = start_A
4   } else {
        i = start_B
6   }

8   if(i == null){
        println("Coda vuota!")
    }
```

```

10     return start_A, start_B
11 }
12
13 while(true){
14     if(sched == 'a'){
15         next = i.next_A
16     } else {
17         next = i.next_B
18     }
19     if(next == null){
20         break
21     }
22     i = next
23 }
24
25 start_A, start_B = run(i, start_A, start_B)
26
27 return start_A, start_B
28 }

```

La funzione che implementa il secondo comando, `run_first()`, prende in input la politica di scheduling attuale ed i due puntatori d'inizio, per poi restituire i due puntatori d'inizio, eventualmente aggiornati, dopo aver eseguito il task in cima alla coda. Questa funzione utilizza un puntatore, `i`, per scorrere la coda, a seconda della politica attualmente selezionata. Il puntatore `i` viene inizializzato al puntatore d'inizio corrispondente alla politica attuale, quindi viene effettuato un ciclo all'interno del quale si scorrono tutti i task della lista (A o B, a seconda della politica di scheduling) per fermarsi soltanto una volta trovato il primo task che non ha successore, ovvero l'ultimo della lista. Avendo implementato le code "al contrario", l'elemento in fondo alla lista corrisponde all'elemento in testa alla coda, ovvero il task che vogliamo eseguire. Una volta individuato il task, si invoca quindi una funzione ausiliare, `run()`, che si occupa di eseguire il task specificato come parametro e restituire i puntatori d'inizio aggiornati. Infine, `run_first()` potrà restituire i nuovi puntatori d'inizio.

La funzione ausiliaria `run()` è una funzione che serve ad eseguire un task specifico, aggiornando le liste di conseguenza. È stato utile scrivere `run()` come funzione a sé stante in quanto essa verrà riutilizzata per l'implementazione del terzo comando, ovvero l'esecuzione di un task specifico. Infatti, pensandoci bene, il secondo e terzo comando fanno più o meno la stessa cosa (eseguire un task): l'unica cosa che cambia è come viene individuato il task da eseguire, ma il resto delle operazioni rimane identico. Quindi si è deciso di implementare l'esecuzione di un task generico come funzione ausiliaria a parte (`run()`, appunto) e di ridurre le implementazioni dei comandi secondo e terzo alla semplice individuazione del task da eseguire (uno cercando quello in testa, l'altro cercando quello con un ID specifico), per poi richiamare entrambi la funzione `run()` sul task individuato.

Lo pseudocodice dell'implementazione di `run()` è mostrata di seguito:

```

run(task, start_A, start_B){
2     task.cycles = task.cycles - 1
3     start_A, start_B = detach(task, start_A, start_B)
4     if(task.cycles > 0){
5         start_A, start_B = insert(task, start_A, start_B)
6     }
7     return start_A, start_B
8 }

```

Quello che fa `run()` è molto semplice. Innanzitutto aggiorna il numero di esecuzioni rimanenti del task, decrementandolo. Quindi invoca una funzione ausiliaria, `detach()`, che serve a “staccare” il task selezionato da entrambe le liste. A questo punto, se le esecuzioni rimanenti non hanno raggiunto lo 0, il task viene reinserito nelle liste, eventualmente in una posizione diversa da quella precedente. Questa tecnica, rimuovere un task e reinserirlo, permette di riutilizzare in maniera intelligente il codice scritto per `insert()` in modo da mantenere le liste sempre aggiornate ogni qual volta si effettua una modifica ad un task (in particolare, alla sua priorità o alle sue esecuzioni rimanenti). Nel caso in cui le esecuzioni rimanenti abbiano raggiunto lo 0, il task verrà quindi eliminato a livello logico dalle liste, ovvero la memoria allocata per il record del task non verrà deallocata, ma verranno semplicemente eliminati i collegamenti al task da entrambe le liste, rendendolo di fatto irraggiungibile e, quindi, come se non esistesse.

Vediamo l’implementazione di `detach()` :

```

detach(task, start_A, start_B){
2   if(start_A.next_A == null){
        start_A = null
4       start_B = null
    } else {
6       prev = task.prev_A
        next = task.next_A
8       if(prev == null){
            start_A = next
10            next.prev_A = null
        } else if(next == null){
12            prev.next_A = null
        } else {
14            prev.next_A = next
            next.prev_A = prev
16        }

18        prev = task.prev_B
        next = task.next_B
20        if(prev == null){
            start_B = next
22            next.prev_B = null
        } else if(next == null){
24            prev.next_B = null
        } else {
26            prev.next_B = next
            next.prev_B = prev
28        }
    }

30    task.prev_A = null
    task.next_A = null
    task.prev_B = null
    task.next_B = null

36    return start_A, start_B
}

```

Quello che fa la funzione ausiliaria `detach()` è molto semplice. Innanzitutto controlla se l’elemento da rimuovere è l’unico della lista, nel qual caso imposta semplicemente i puntatori d’inizio a `null`. Altrimenti, carica il task precedente ed il successivo del task da rimuovere (prima nella lista A, poi nella B) e li collega tra loro, facendo anche controlli nel caso in cui il

task da rimuovere fosse stato il primo (nel qual caso aggiorna anche il puntatore d'inizio) oppure l'ultimo. Infine imposta tutti i puntatori del task a `null` (quest'ultima operazione non sarebbe necessaria, viene fatta più per un motivo correttezza e consistenza).

Con `detach()` abbiamo terminato la descrizione delle funzioni ausiliarie implementate nel codice del programma, quindi possiamo passare a descrivere le funzioni dei comandi rimanenti. Di seguito, la funzione `run_id()`, che implementa il comando per l'esecuzione di un task specifico:

```
run_id(start_A, start_B){
2   if(start_A == null){
      println("Coda vuota!")
4   return start_A, start_B
    }

6   while(true){
8     id = read_int("Inserire l'ID del task da eseguire: ")
      task = find_id(id, start_A)
10    if(task == null){
          println("Task con ID " + id + "non trovato.")
12    } else {
          break
14    }
    }

16    start_A, start_B = run(task, start_A, start_B)

18    return start_A, start_B
20 }
```

Vedendo la descrizione di `run_id()` possiamo subito notare come l'aver implementato le funzioni ausiliarie di cui abbiamo parlato prima abbia semplificato enormemente l'implementazione di tutte le altre funzioni. `run_id()`, infatti, si limita a chiedere all'utente un ID da linea di comando finché non viene trovato nelle liste un task con l'ID specificato (funzione ausiliaria `find_id()`). Quindi, una volta recuperato il task che si vuole eseguire, basterà passarlo alla funzione ausiliaria `run()` per ottenere i risultati prefissati.

Di seguito, la descrizione ad alto livello della funzione per l'eliminazione di un task con ID specificato dalla coda:

```
delete_id(start_A, start_B){
2   if(start_A == null){
      println("Coda vuota!")
4   return start_A, start_B
    }

6   while(true){
8     id = read_int("Inserire l'ID del task da eliminare: ")
      task = find_id(id, start_A)
10    if(task == null){
          println("Task con ID " + id + "non trovato.")
12    } else {
          break
14    }
    }

16    start_A, start_B = detach(task, start_A, start_B)

18 }
```

```

20  }
    return start_A, start_B

```

Nuovamente, possiamo vedere quanto le funzioni ausiliarie che abbiamo incluso rendano l'implementazione più semplice. La funzione `delete_id()`, infatti, è strutturata esattamente come `run_id()`, con l'unica differenza che, una volta trovato il task con ID specificato da input, anziché passarlo come parametro a `run()` lo passa come parametro a `detach()`, ottenendo, di fatto, l'eliminazione logica del task da entrambe le liste.

Il prossimo frammento di pseudocodice rappresenta invece l'implementazione della funzione `change_prio()`, corrispondente al comando 5, che cambia la priorità di un task con ID specificato:

```

change_prio(start_A, start_B){
2   if(start_A == null){
        println("Coda vuota!")
4       return start_A, start_B
    }

6
    while(true){
8        id = read_int("Inserire l'ID del task da modificare: ")
        task = find_id(id, start_A)
10       if(task == null){
            println("Task con ID " + id + "non trovato.")
12        } else {
            break
14        }
    }

16    while(true){
        prio = read_int("Inserire la nuova priorità: ")
18        if(prio >= 0 AND prio <= 9){
            task.prio = prio
20            break
        }

22    }

24    start_A, start_B = detach(task, start_A, start_B)
    start_A, start_B = insert(task, start_A, start_B)

26
    return start_A, start_B
28 }

```

Essendo `change_prio()` una funzione che, come le due precedenti, si basa sull'eseguire una certa operazione su un task con ID specificato, rispetto alle funzioni `run_id()` e `delete_id()` cambia soltanto la parte centrale, che rappresenta l'operazione eseguita sul task individuato. In questo caso, una volta individuato il task richiesto, viene chiesto all'utente di inserire una priorità (compresa tra 0 e 9, come visto prima in `insert_task()`) che verrà impostata come nuova priorità del task. È necessario, quindi, aggiornare la posizione del task nelle due liste, in quanto la modifica della priorità potrebbe aver cambiato l'ordine relativo dei task. Per fare questo, si rimuove il task dalle liste, invocando `detach()`, e quindi lo si reinserisce invocando `insert()`, che si occuperà di inserirlo nella posizione corretta in entrambe le liste tenendo conto della priorità aggiornata.

Infine, vediamo l'implementazione della funzione che permette di passare da una politica di scheduling all'altra:



```

change_sched(old_sched){
2   if(old_sched == 'a'){
        new_sched = 'b'
4   } else {
        new_sched = 'a'
6   }
    return new_sched
8 }

```

Quest'ultima funzione, molto semplice, controlla semplicemente qual è la politica di scheduling attuale e restituisce l'altra politica, la quale, all'interno del `main()`, verrà salvata come politica di scheduling attuale.

## Motivazione delle scelte implementative

Discutiamo, adesso, alcune scelte implementative fatte.

La scelta implementativa forse più importate è quella di aver implementato le code tramite liste mantenute sempre ordinate. La principale alternativa, in questo senso, sarebbe stata quella di mantenere una sola lista, senza un ordine preciso, andando a ricavare l'ordine corretto ogni qual volta venga invocata una funzione per la quale l'ordine è discriminante, come l'esecuzione del primo task o la stampa della coda. Se avessimo implementato questa alternativa, alcune funzioni sarebbero risultate molto più semplici: la funzione `insert()`, per essere più specifici, sarebbe costata un  $\mathcal{O}(1)$ , in quanto sarebbe bastato inserire il task in una posizione qualsiasi (ad esempio la prima) della lista. Di conseguenza, tutte le funzioni che invocano `insert()`, come `insert_task()` o `run()`, sarebbero a loro volta risultate molto più semplici, sia a livello implementativo che computazionale, non dovendo preoccuparsi di mantenere l'ordine nella lista. Il problema con questa implementazione sorge al momento in cui è necessario ricavare l'ordine relativo dei task, ovvero durante la funzione `run_first()` (in quanto è necessario ricavare il task in testa alla coda) e durante la stampa della coda (in quanto la coda dev'essere stampata in ordine). Sappiamo che ordinare una lista di  $n$  elementi costa almeno un  $\mathcal{O}(n \log_2 n)$ , anche se questo comporterebbe dover implementare in linguaggio assembler algoritmi di ordinamento complessi come il Quicksort od il Mergesort. Altrimenti si potrebbe implementare un algoritmo di ordinamento più semplice, come l'Insertionsort, che però ha un costo dell'ordine di  $\mathcal{O}(n^2)$ . Quindi, con questa implementazione, ogni volta che si vuole eseguire il comando in testa alla coda, è necessario fare  $\mathcal{O}(n^2)$  operazioni. Inoltre, verrebbero fatte  $\mathcal{O}(n^2)$  operazioni anche ogni volta che viene stampata la coda, ovvero dopo l'esecuzione di un qualsiasi comando.

La scelta implementativa che è stata fatta in questo caso si basa quindi sul concetto di mantenere una lista ordinata dopo ogni operazione. Così facendo, le operazioni che coinvolgono `insert()` saranno certamente più complesse, ma altre operazioni risulteranno essere più leggere, andando a far calare il costo computazionale complessivo. In particolare, con l'implementazione presentata, ogni chiamata ad `insert()` costa  $\mathcal{O}(n)$ , in quanto è necessario scorrere la lista al più una volta per trovare il punto in cui inserire il task richiesto. Il vantaggio sta proprio nel fatto che l'operazione implementata da `insert()` non è un vero e proprio ordinamento, ma l'inserimento di un nuovo elemento all'interno di una lista già ordinata! Questo rappresenta un grosso vantaggio rispetto a dover ordinare la lista ogni volta. Infatti in questo modo per eseguire il task in testa alla coda basterà scorrere tutta la lista fino all'ultimo elemento, che costa sempre  $\mathcal{O}(n)$  (implementando la coda in senso corretto sarebbe costato  $\mathcal{O}(1)$ , ma di questo parleremo poco più avanti). Inoltre, anche la stampa della coda ha un costo lineare  $\mathcal{O}(n)$ , in quanto gli elementi vengono scorsi dal primo (fondo della coda) all'ultimo (testa). Per quanto riguarda le altre operazioni, troviamo operazioni molto semplici, come `detach()` o `change_sched()`, che

hanno un costo costante  $\mathcal{O}(1)$ , mentre le altre operazioni, come ad esempio `find_id()`, costano sempre  $\mathcal{O}(n)$ .

Ricapitolando, il limite superiore asintotico del costo computazionale della soluzione implementata rimane un  $\mathcal{O}(n)$ , mentre con la soluzione alternativa diventa un  $\mathcal{O}(n^2)$  (o anche  $\mathcal{O}(n \log_2 n)$  nel caso ottimistico in cui si riesce ad implementare un algoritmo più complesso in assembler).

A questo punto ci si potrebbe chiedere come mai mantenere due liste contemporaneamente anziché una soltanto alla volta. Dalle considerazioni appena fatte la risposta è chiara. Mantenere una sola lista implica che al cambio della politica di scheduling è necessario ordinare da zero l'intera lista che, come visto prima, costa  $\mathcal{O}(n^2)$ . Inoltre, con questa scelta, l' `insert()` continuerebbe ad avere un costo  $\mathcal{O}(n)$  e non costante. Quindi questa scelta implementativa è la peggiore, in quanto unirebbe il peggio di entrambe le alternative.

L'ultimo aspetto da analizzare è il perché si è scelto di implementare la coda al contrario anziché nel senso naturale. La scelta è nata dall'osservazione che, durante l'esecuzione del programma, in media sono più le stampe della coda (una ogni comando eseguito) rispetto alle esecuzioni del task in testa alla coda. Questo ci suggerisce che, se c'è da scegliere quale operazione rendere più veloce, la scelta più logica è prediligere l'operazione di stampa. Le due operazioni, infatti, hanno bisogno degli estremi opposti della coda: la stampa deve partire dal fondo, mentre `run_first()` ha bisogno della testa. Implementando la coda al contrario si predilige quindi la stampa della coda, che quindi impiega  $\mathcal{O}(n)$  passi, anziché  $\mathcal{O}(2n)$ . Certo in questo modo `run_first()` viene a costare  $\mathcal{O}(n)$  anziché  $\mathcal{O}(1)$ , tuttavia ricordiamo che dopo `run_first()` deve comunque essere stampata la coda, portando ad un costo complessivo di  $\mathcal{O}(2n)$  in ogni caso. Quindi la scelta si riduce a voler eseguire  $\mathcal{O}(2n)$  passi soltanto quando si esegue `run_first()` oppure sempre. Chiaramente la scelta più performante è la prima, che si traduce nell'implementazione al contrario della coda.

Si sarebbero potuti, in alternativa, mantenere anche dei puntatori alla testa, in aggiunta a quelli già presenti, ma per non appesantire troppo l'implementazione e cercare di rendere il codice il più chiaro e leggibile possibile abbiamo ritenuto che mantenere due puntatori era già abbastanza e che mantenerne quattro sarebbe stato troppo complicato e caotico, anche se avrebbe portato certamente ad un incremento delle prestazioni del programma.

## Uso di registri e memoria

Per quanto riguarda l'uso dei registri, valgono le stesse considerazioni fatte per l'esercizio precedente, in quanto le convenzioni sono sempre state rispettate, utilizzando di volta in volta i registri più adatti. Una convenzione adottata in questo esercizio è che generalmente si è assegnato il registro `$s5` alla politica di scheduling, `$s6` al puntatore d'inizio A e `$s7` al puntatore d'inizio B, in quanto questi sono registri ricorrenti ed utilizzare gli ultimi ci ha permesso di non fare confusione con altri registri utilizzati in modo diverso di volta in volta. All'interno delle procedure che non prevedono di preservare questi tre valori (perché sono procedure foglia oppure perché ne calcolano una versione aggiornata) i registri utilizzati saranno invece, rispettivamente, `$t7`, `$t8` e `$t9`, per le stesse motivazioni.

Lo stack frame viene utilizzato, come sempre, per memorizzare i dati di registri importanti prima di ogni chiamata a procedura, per poi poterli recuperare in un secondo momento.

L'heap viene allocato dinamicamente ogni volta che viene creato un nuovo task, utilizzando la chiamata di sistema `SBRK`. In particolare ogni creazione di un nuovo task comporta l'allocazione in memoria dinamica di un totale di 36 byte (o 9 word) così strutturati: 4 byte (range 0-4) per il puntatore al precedente A, 4 byte (range 4-8) per il puntatore al precedente B, 4 byte (range 8-12) per l'ID del task, 8 byte (range 12-20) per il nome del task, 4 byte (range 20-24) per la priorità del task, 4 byte (range 24-28) per le esecuzioni rimanenti del task, 4 byte (range 28-32) per il puntatore al successivo A e 4 byte (range 32-36) per il puntatore al successivo B. Mentre la

memoria allocata all'interno dello stack frame viene deallocata quando non serve più, la memoria allocata all'interno dell'heap non viene deallocata, ma soltanto eliminata a livello logico (come visto prima).

L'evoluzione tipica dello stack frame vede la chiamata ad una funzione corrispondente ad uno dei comandi disponibili, come `insert_task()` o `run_id()`, seguita da una o più funzioni ausiliarie, a seconda del comando scelto. In ogni caso, la profondità delle chiamate non è virtualmente infinita, come nel caso dell'esercizio precedente, ma si limita, solo per alcune operazioni, a poche chiamate annidate. Questo perché in questo esercizio non è presente ricorsione, mentre nel precedente sì (anche se non diretta).

## Simulazioni

Mostriamo adesso il comportamento del programma, tramite una serie di immagini in sequenza, tentando di coprire il più possibile tutti i casi possibili.

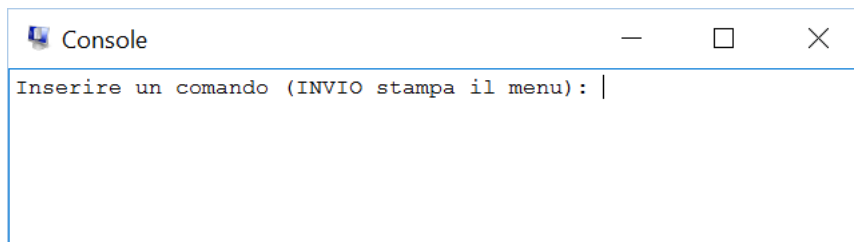


Figura 4: Messaggio iniziale su consolle.

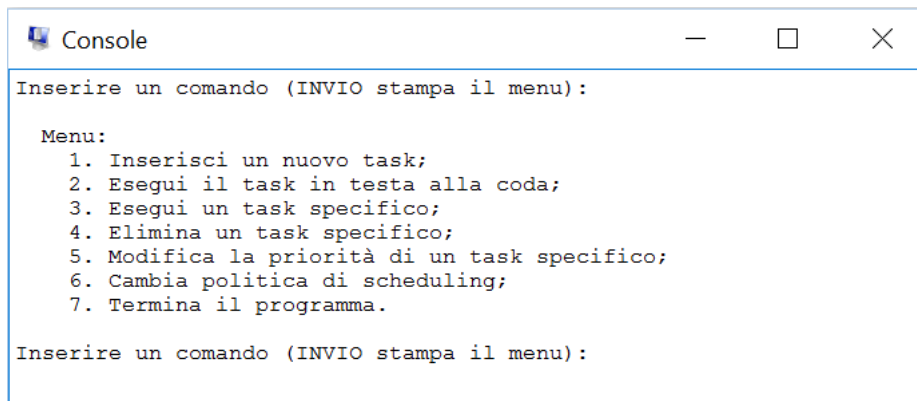


Figura 5: Premendo Invio, viene stampato il menu del programma.



```

Console
Inserire un comando (INVIO stampa il menu): 5

Modifica della priorità di un task specifico.
Coda vuota.
Nessun task modificato!

Scheduling su PRIORITÀ
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
|                                     |
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |

```

Figura 9: Analogamente per la modifica della priorità di un task specifico.

```

Console
Inserire un comando (INVIO stampa il menu): 6

Cambio della politica di scheduling.
Politica di scheduling attuale: su PRIORITÀ
Nuova politica di scheduling: su ESECUZIONI RIMANENTI
Politica di scheduling correttamente cambiata!

Scheduling su ESECUZIONI RIMANENTI
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
|                                     |
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |

```

Figura 10: Il cambio di politica di scheduling viene eseguito con successo e la tabella (vuota) stampata.

```

Console
Inserire un comando (INVIO stampa il menu): 1

Inserimento di un nuovo task.
Inserire l'ID del task: 1
Inserire il nome del task (max 8 caratteri): mips
Inserire la priorità del task (min 0, max 9): 10
Inserire la priorità del task (min 0, max 9): 2
Inserire il numero di cicli di CPU del task (min 1, max 99): 0
Inserire il numero di cicli di CPU del task (min 1, max 99): 14
Task correttamente inserito!

Scheduling su ESECUZIONI RIMANENTI
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
| 1 / 2 / mips / 14 / |
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |

```

Figura 11: Iniziamo ad inserire task nella coda: i dati vengono chiesti uno dopo l'altro e nel caso siano fuori dal range previsto vengono chiesti nuovamente.

```
Console
Inserire un comando (INVIO stampa il menu): 1

Inserimento di un nuovo task.
  Inserire l'ID del task: 1
  Task con ID 1 già presente.
  Inserire l'ID del task: 2
  Inserire il nome del task (max 8 caratteri): testtask
  Inserire la priorità del task (min 0, max 9): 4
  Inserire il numero di cicli di CPU del task (min 1, max 99): 9
Task correttamente inserito!

Scheduling su ESECUZIONI RIMANENTI
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/  2 /      4   / testtask /          9   /
+---+-----+-----+-----+
/  1 /      2   / mips    /         14   /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |
```

Figura 12: Inseriamo un secondo elemento: vediamo che scegliere un ID già riservato comporta la richiesta di un nuovo ID.

```
Console
Inserire un comando (INVIO stampa il menu): 1

Inserimento di un nuovo task.
  Inserire l'ID del task: 3
  Inserire il nome del task (max 8 caratteri): lol
  Inserire la priorità del task (min 0, max 9): 3
  Inserire il numero di cicli di CPU del task (min 1, max 99): 70
Task correttamente inserito!

Scheduling su ESECUZIONI RIMANENTI
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/  2 /      4   / testtask /          9   /
+---+-----+-----+-----+
/  1 /      2   / mips    /         14   /
+---+-----+-----+-----+
/  3 /      3   / lol     /         70   /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):
```

Figura 13: Inseriamo un terzo elemento.

```
Console
Inserire un comando (INVIO stampa il menu): 1

Inserimento di un nuovo task.
  Inserire l'ID del task: 4
  Inserire il nome del task (max 8 caratteri): asd
  Inserire la priorità del task (min 0, max 9): 1
  Inserire il numero di cicli di CPU del task (min 1, max 99): 9
Task correttamente inserito!

Scheduling su ESECUZIONI RIMANENTI
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+-----+-----+-----+-----+
/ 2 /      4 / testtask /          9 /
+-----+-----+-----+-----+
/ 4 /      1 / asd      /          9 /
+-----+-----+-----+-----+
/ 1 /      2 / mips     /         14 /
+-----+-----+-----+-----+
/ 3 /      3 / lol      /         70 /
+-----+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):
```

Figura 14: Inseriamo un task con stesse esecuzioni rimanenti del task *testtask* ma priorità minore: il nuovo task risulta correttamente più avanti nella coda.

```
Console
Inserire un comando (INVIO stampa il menu): 2

Esecuzione del primo task.
  Task con ID 3 eseguito.
Task correttamente eseguito!

Scheduling su ESECUZIONI RIMANENTI
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+-----+-----+-----+-----+
/ 2 /      4 / testtask /          9 /
+-----+-----+-----+-----+
/ 4 /      1 / asd      /          9 /
+-----+-----+-----+-----+
/ 1 /      2 / mips     /         14 /
+-----+-----+-----+-----+
/ 3 /      3 / lol      /         69 /
+-----+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):
```

Figura 15: Eseguiamo il task in testa alla coda: le esecuzioni rimanenti di *lol* sono scese da 70 a 69.

```

Console
Inserire un comando (INVIO stampa il menu): 6

Cambio della politica di scheduling.
  Politica di scheduling attuale: su ESECUZIONI RIMANENTI
  Nuova politica di scheduling: su PRIORITÀ
  Politica di scheduling correttamente cambiata!

Scheduling su PRIORITÀ
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/ 2 /      4   / testtask /      9   /
+---+-----+-----+-----+
/ 3 /      3   / lol      /     69   /
+---+-----+-----+-----+
/ 1 /      2   / mips      /     14   /
+---+-----+-----+-----+
/ 4 /      1   / asd       /      9   /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):

```

Figura 16: Cambiamo la politica di scheduling di nuovo: la coda viene aggiornata di conseguenza rispettando la priorità.

```

Console
Inserire un comando (INVIO stampa il menu): 1

Inserimento di un nuovo task.
  Inserire l'ID del task: 5
  Inserire il nome del task (max 8 caratteri): kek
  Inserire la priorità del task (min 0, max 9): 2
  Inserire il numero di cicli di CPU del task (min 1, max 99): 1
Task correttamente inserito!

Scheduling su PRIORITÀ
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/ 2 /      4   / testtask /      9   /
+---+-----+-----+-----+
/ 3 /      3   / lol      /     69   /
+---+-----+-----+-----+
/ 5 /      2   / kek       /      1   /
+---+-----+-----+-----+
/ 1 /      2   / mips      /     14   /
+---+-----+-----+-----+
/ 4 /      1   / asd       /      9   /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):

```

Figura 17: Inseriamo un task con stessa priorità di *mips* ma meno esecuzioni rimanenti: come prima, viene inserito correttamente prima di *mips*.



```
Console
Inserire un comando (INVIO stampa il menu): 5

Modifica della priorità di un task specifico.
Inserire l'ID del task da modificare: 3
Inserire la nuova priorità del task (min 0, max 9): 1
Priorità del task correttamente modificata!

Scheduling su PRIORITÀ
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/ 2 /      4 / testtask /          9 /
+---+-----+-----+-----+
/ 5 /      2 / kek      /          1 /
+---+-----+-----+-----+
/ 1 /      2 / mips     /         14 /
+---+-----+-----+-----+
/ 4 /      1 / asd      /          9 /
+---+-----+-----+-----+
/ 3 /      1 / lol      /         69 /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):
```

Figura 18: Modifichiamo la priorità del task *lol*: il task viene correttamente spostato.

```
Console
Inserire un comando (INVIO stampa il menu): 4

Eliminazione di un task specifico.
Inserire l'ID del task da eliminare: 2
Task correttamente eliminato!

Scheduling su PRIORITÀ
+---+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+---+-----+-----+-----+
/ 5 /      2 / kek      /          1 /
+---+-----+-----+-----+
/ 1 /      2 / mips     /         14 /
+---+-----+-----+-----+
/ 4 /      1 / asd      /          9 /
+---+-----+-----+-----+
/ 3 /      1 / lol      /         69 /
+---+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |
```

Figura 19: Eliminiamo il task *testtask*.

```
Console
Inserire un comando (INVIO stampa il menu): 3

Esecuzione di un task specifico.
  Inserire l'ID del task da eseguire: 5
  Task con ID 5 terminato!
  Task correttamente eseguito!

Scheduling su PRIORITÀ
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+-----+-----+-----+-----+
/  1 /      2   / mips   /      14   /
+-----+-----+-----+-----+
/  4 /      1   / asd    /       9   /
+-----+-----+-----+-----+
/  3 /      1   / lol    /      69   /
+-----+-----+-----+-----+

Inserire un comando (INVIO stampa il menu): |
```

Figura 20: Eseguiamo il task *kek* specificandone l'ID: il task aveva 1 esecuzione rimanente, quindi viene rimosso dalla coda.

```
Console
Inserire un comando (INVIO stampa il menu): 6

Cambio della politica di scheduling.
  Politica di scheduling attuale: su PRIORITÀ
  Nuova politica di scheduling: su ESECUZIONI RIMANENTI
  Politica di scheduling correttamente cambiata!

Scheduling su ESECUZIONI RIMANENTI
+-----+-----+-----+-----+
| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |
+-----+-----+-----+-----+
/  4 /      1   / asd    /       9   /
+-----+-----+-----+-----+
/  1 /      2   / mips   /      14   /
+-----+-----+-----+-----+
/  3 /      1   / lol    /      69   /
+-----+-----+-----+-----+

Inserire un comando (INVIO stampa il menu):
```

Figura 21: Cambiamo politica di scheduling un'ultima volta.

```
Console
Inserire un comando (INVIO stampa il menu): 7

Programma terminato.
```

Figura 22: Infine si termina il programma.

## Codice MIPS

Di seguito, il codice MIPS completo che implementa il programma descritto dal secondo esercizio, opportunamente commentato.

```
1  # Title: Scheduler di processi                               Filename: es2.s
2  # Author1: ??? ??????   ????????   ????.?????@stud.unifi.it
3  # Author2: ??? ??????   ????????   ????.?????@stud.unifi.it
4  # Author3: ??? ??????   ????????   ????.?????@stud.unifi.it
5  # Date: ??/??/????
6  # Description: Gestione dello scheduling di processi
7  # Input: Comandi previsti (da 1 a 7) tramite linea di comando
8  # Output: Log dell'esecuzione dei comandi scelti e coda aggiornata
9
10 ##### Data segment #####
11 .data
12 buffer:                .space 1024
13 tab:                   .asciiz "\t"
14 newline:               .asciiz "\n"
15 insert_command:        .asciiz "Inserire un comando (INVIO stampa il menu): "
16 menu:                  .asciiz "  Menu:\n    1. Inserisci un nuovo task;\n    2.
    Esegui il task in testa alla coda;\n    3. Esegui un task specifico;\n    4.
    Elimina un task specifico;\n    5. Modifica la priorità di un task specifico;\n
    6. Cambia politica di scheduling;\n    7. Termina il programma.\n\n"
17 exiting:               .asciiz "  Programma terminato."
18 insert_task_msg:       .asciiz "  Inserimento di un nuovo task.\n"
19 insert_id_msg:         .asciiz "    Inserire l'ID del task: "
20 insert_name_msg:       .asciiz "    Inserire il nome del task (max 8 caratteri): "
21 insert_prio_msg:       .asciiz "    Inserire la priorità del task (min 0, max 9): "
22 insert_cycles_msg:     .asciiz "    Inserire il numero di cicli di CPU del task (min
    1, max 99): "
23 insert_task_done_msg:  .asciiz "  Task correttamente inserito!\n\n"
24 run_first_msg:         .asciiz "  Esecuzione del primo task.\n"
25 empty_msg:             .asciiz "    Coda vuota.\n"
26 task_msg:              .asciiz "    Task con ID "
27 executed_msg:         .asciiz "  eseguito.\n"
28 completed_msg:        .asciiz "  terminato!\n"
29 run_done_msg:          .asciiz "  Task correttamente eseguito!\n\n"
30 run_not_done_msg:      .asciiz "  Nessun task eseguito!\n\n"
31 run_id_msg:            .asciiz "  Esecuzione di un task specifico.\n"
32 choose_id_msg:         .asciiz "    Inserire l'ID del task da "
33 choose_id_run_msg:     .asciiz "  eseguire: "
34 id_not_found_msg:      .asciiz "  non trovato.\n"
35 id_duplicate_msg:      .asciiz "  già presente.\n"
36 delete_id_msg:         .asciiz "  Eliminazione di un task specifico.\n"
37 choose_id_delete_msg:  .asciiz "  eliminare: "
38 new_prio_msg:          .asciiz "    Inserire la nuova priorità del task (min 0, max
    9): "
39 delete_id_done_msg:    .asciiz "  Task correttamente eliminato!\n\n"
40 delete_id_not_done_msg: .asciiz "  Nessun task eliminato!\n\n"
41 change_prio_msg:       .asciiz "  Modifica della priorità di un task specifico.\n"
42 choose_id_change_prio_msg: .asciiz "  modificare: "
43 change_prio_done_msg:  .asciiz "  Priorità del task correttamente modificata!\n\n"
44 change_prio_not_done_msg: .asciiz "  Nessun task modificato!\n\n"
45 change_sched_msg:      .asciiz "  Cambio della politica di scheduling.\n"
46 change_sched_old_msg:  .asciiz "    Politica di scheduling attuale: su "
47 change_sched_new_msg:  .asciiz "    Nuova politica di scheduling: su "
```

```

48 change_sched_done_msg:      .ascii "  Politica di scheduling correttamente cambiata!\n\n"
49 scheduling_msg:             .ascii "  Scheduling su "
50 sched_prio_msg:             .ascii "PRIORITÀ\n"
51 sched_cycles_msg:           .ascii "ESECUZIONI RIMANENTI\n"
52 print_delim:                .ascii "  +---+-----+-----+-----+-----+\n"
53 print_header:               .ascii "  | ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |\n"
54 print_empty_queue:          .ascii "  |                                     Coda vuota!                |\n"
55 print_row_start:            .ascii "  /\n"
56 print_row_five_spaces:      .ascii "      "\n"
57 print_row_eight_spaces:     .ascii "          "\n"
58
59 ##### Code segment #####
60 .text
61 .globl main
62
63 ### insert ###
64 insert :                    # procedura per l'inserimento di un task nelle liste
65     move $t0, $a0           # primo parametro: puntatore al task da inserire
66     move $t8, $a1           # secondo: puntatore alla lista A
67     move $t9, $a2           # terzo: puntatore alla lista B
68
69     beq $t8, $zero, insert_first # se il puntatore d'inizio è nullo (coda vuota), salta
all'inserimento del primo task nella coda
70
71     lw $t1, 20($t0)          # carica la priorità
72     lw $t2, 24($t0)          # e le esecuzioni rimanenti del task da inserire
73
74     move $t3, $t8            # carica il puntatore d'inizio della lista A
75
76 insert_prio_loop:
77     lw $t4, 20($t3)          # carica la priorità del task attuale
78     bgt $t4, $t1, insert_prio_next # se la il task attuale ha priorità maggiore
del task da inserire, passa al prossimo task
79     beq $t4, $t1, insert_prio_cycles_loop # se la priorità è uguale, inizia a scorrere
comparando le esecuzioni rimanenti
80     j insert_prio_found      # altrimenti, si è trovato il punto in cui
inserire il task
81
82 insert_prio_next:           # istruzioni per passare al prossimo task della
lista A
83     lw $t4, 28($t3)          # carica il puntatore al prossimo task della lista
A
84     beq $t4, $zero, insert_prio_last # se il puntatore è nullo, il task è da inserire
come ultimo della lista
85     move $t3, $t4            # altrimenti, passa al prossimo
86     j insert_prio_loop       # ed esegue un altro ciclo del loop
87
88 insert_prio_cycles_loop:    # ciclo per la comparazione su esecuzioni
rimanenti sulla lista A
89     lw $t4, 20($t3)          # carica la priorità del task attuale
90     blt $t4, $t1, insert_prio_found # se la priorità è minore, la sfilza di task
con priorità uguale è terminata e si è trovato dove inserire il task
91     lw $t4, 24($t3)          # altrimenti, carica le esecuzioni rimanenti
del task attuale

```

```

92     blt $t4, $t2, insert_prio_cycles_next # se le esecuzioni rimanenti del task attuale
sono minori (strettamente) del task da inserire, continua a cercare comparando le
esec. rimanenti
93     j insert_prio_found                  # altrimenti, si è trovato dove inserire il
task
94
95 insert_prio_cycles_next:
96     lw $t4, 28($t3)                    # carica il prossimo elemento della lista A
97     beq $t4, $zero, insert_prio_last   # se il puntatore è nullo, inserisce il task come
ultimo della lista
98     move $t3, $t4                      # altrimenti, passa al prossimo
99     j insert_prio_cycles_loop          # ed esegue un altro ciclo (comparando su
esecuzioni rimanenti)
100
101 insert_prio_found:                    # quando il task è stato trovato
102     sw $t3, 28($t0)                    # salva il task attuale come prossimo del task da
inserire
103     lw $t4, 0($t3)                    # carica il task precedente al task attuale
104     sw $t0, 0($t3)                    # imposta il task da inserire come nuovo
precedente del task attuale
105     beq $t4, $zero, insert_prio_first # se il vecchio precedente è nullo, allora il task
è da inserire come primo della lista
106     sw $t4, 0($t0)                    # altrimenti, imposta il task precedente al task
attuale come precedente del task da inserire
107     sw $t0, 28($t4)                    # ed imposta il task da inserire come successivo
del precedente del task attuale
108     j insert_cycles                    # infine salta alle istruzioni per l'inserimento
nella lista B
109
110 insert_prio_first :                    # se il task è da inserire come primo della lista
111     sw $zero, 0($t0)                  # salva un puntatore nullo come precedente del task da inserire
112     move $t8, $t0                     # ed imposta il puntatore d'inizio della lista A al task da
inserire
113     j insert_cycles                    # quindi passa alla lista B
114
115 insert_prio_last :                    # se il task è da inserire come ultimo della lista
116     sw $t0, 28($t3)                  # salva il task da inserire come successivo del task attuale
117     sw $t3, 0($t0)                   # salva il task attuale come precedente del task da inserire
118     sw $zero, 28($t0)                 # ed salva un puntatore nullo come successivo del task da
inserire
119
120 insert_cycles :                        # le istruzioni per l'inserimento nella lista B sono speculari a
quelle per la lista A
121     move $t3, $t9
122
123 insert_cycles_loop:
124     lw $t4, 24($t3)
125     blt $t4, $t2, insert_cycles_next
126     beq $t4, $t2, insert_cycles_prio_loop
127     j insert_cycles_found
128
129 insert_cycles_next:
130     lw $t4, 32($t3)
131     beq $t4, $zero, insert_cycles_last
132     move $t3, $t4
133     j insert_cycles_loop
134
135 insert_cycles_prio_loop:

```

```

136     lw $t4, 24($t3)
137     bgt $t4, $t2, insert_cycles_found
138     lw $t4, 20($t3)
139     bgt $t4, $t1, insert_cycles_prio_next
140     j insert_cycles_found
141
142 insert_cycles_prio_next:
143     lw $t4, 32($t3)
144     beq $t4, $zero, insert_cycles_last
145     move $t3, $t4
146     j insert_cycles_prio_loop
147
148 insert_cycles_found:
149     sw $t3, 32($t0)
150     lw $t4, 4($t3)
151     sw $t0, 4($t3)
152     beq $t4, $zero, insert_cycles_first
153     sw $t4, 4($t0)
154     sw $t0, 32($t4)
155     j insert_done
156
157 insert_cycles_first :
158     sw $zero, 4($t0)
159     move $t9, $t0
160     j insert_done
161
162 insert_cycles_last :
163     sw $t0, 32($t3)
164     sw $t3, 4($t0)
165     sw $zero, 32($t0)
166     j insert_done
167
168 insert_first :      # se la coda è vuota
169     move $t8, $t0    # fa puntare entrambi i puntatori d'inizio (delle due liste)
170     move $t9, $t0    # al task da inserire
171
172 insert_done:        # al termine dell'inserimento
173     move $v0, $t8    # imposta i nuovi puntatori d'inizio delle due liste
174     move $v1, $t9    # come valori di ritorno
175     jr $ra           # quindi, ritorna al chiamante
176 ### insert end ###
177
178 ### detach ###
179 detach:             # procedura per rimuovere un task dalle liste
180     move $t0, $a0    # primo parametro: puntatore al task da rimuovere
181     move $t8, $a1    # secondo: puntatore alla lista A
182     move $t9, $a2    # terzo: puntatore alla lista B
183
184     lw $t1, 28($t8)  # carica il successivo del primo task nella lista
185     beq $t1, $zero, detach_only_task # se è nullo, allora c'è un solo task nella coda (
186                                     # altrimenti
187     lw $t1, 0($t0)    # carica il precedente del task da rimuovere
188     lw $t2, 28($t0)   # ed il successivo
189     beq $t1, $zero, detach_first_prio # se il puntatore al precedente è nullo, allora è
190     da rimuovere il primo task della lista A
191     beq $t2, $zero, detach_last_prio  # se è nullo il puntatore al successivo, allora è

```

```

191     l'ultimo
192     sw $t2, 28($t1)          # altrimenti, imposta il successivo come
    successivo del precedente
193
194     j detach_cycles # salta alla rimozione del task dalla lista B
195
196 detach_first_prio:          # se il task da rimuovere è il primo della lista A
197     move $t8, $t2           # fa puntare il puntatore d'inizio di A al task successivo (
    ovvero il secondo)
198     sw $zero, 0($t2)        # ed imposta il precedente del successivo come puntatore nullo
199
200     j detach_cycles         # quindi salta alla rimozione dalla lista B
201
202 detach_last_prio:          # se il task da rimuovere è l'ultimo della lista A
203     sw $zero, 28($t1)       # basta impostare il successivo del task precedente come
    puntatore nullo
204
205 detach_cycles:             # la rimozione dalla lista B è del tutto analoga a
    quella della lista A
206     lw $t1, 4($t0)
207     lw $t2, 32($t0)
208     beq $t1, $zero, detach_first_cycles
209     beq $t2, $zero, detach_last_cycles
210     sw $t2, 32($t1)
211     sw $t1, 4($t2)
212
213     j detach_done
214
215 detach_first_cycles:
216     move $t9, $t2
217     sw $zero, 4($t2)
218
219     j detach_done
220
221 detach_last_cycles:
222     sw $zero, 32($t1)
223
224     j detach_done
225
226 detach_only_task:          # se il task da rimuovere è l'unico presente nelle due code
227     move $t8, $zero # imposta i due puntatori iniziali delle liste A e B
228     move $t9, $zero # a puntatori nulli (coda vuota)
229
230 detach_done:              # una volta terminata l'eliminazione dalle due liste
231     sw $zero, 0($t0)        # imposta a puntatore nullo il precedente, nella lista A, del
    task rimosso
232     sw $zero, 4($t0)        # il puntatore al precedente nella lista B
233     sw $zero, 28($t0)       # il puntatore al successivo nella lista A
234     sw $zero, 32($t0)       # ed il puntatore al successivo nella lista B
235
236     move $v0, $t8           # imposta il puntatore d'inizio A come primo valore di ritorno
237     move $v1, $t9           # ed il puntatore d'inizio B come secondo valore di ritorno
238     jr $ra                 # torna alla procedura chiamante
239 ### detach end ###
240
241 ### run ###
242 run:                      # procedura per l'esecuzione di un task

```

```

243  move $s0, $a0 # primo parametro: puntatore al task da eseguire
244  move $t8, $a1 # secondo: puntatore alla lista A
245  move $t9, $a2 # terzo: puntatore alla lista B
246
247  lw $s1, 24($s0) # carica le esecuzioni rimanenti del task
248  addi $s1, $s1, -1 # le decrementa
249  sw $s1, 24($s0) # e salva il nuovo valore nel record del task
250
251  addi $sp, $sp, -12 # alloca spazio nello stack per 12 byte (3 word)
252  sw $ra, 8($sp) # salva nello stack l'indirizzo di ritorno
253  sw $s0, 4($sp) # il puntatore al task da eseguire
254  sw $s1, 0($sp) # ed il numero di esecuzioni rimanenti dopo la sua esecuzione
255
256  jal detach # chiama la funzione detach, per rimuovere il task dalle liste
257
258  lw $t0, 4($sp) # recupera dallo stack il puntatore al task da eseguire
259  lw $t1, 0($sp) # e le esecuzioni rimanenti dopo l'esecuzione
260  move $t8, $v0 # come valori di ritorno di detach recupera il puntatore d'inizio A
261  move $t9, $v1 # ed il puntatore d'inizio B
262
263  li $v0, 4 # stampa "Task con ID "
264  la $a0, task_msg
265  syscall
266
267  li $v0, 1 # stampa l'ID del task
268  lw $a0, 8($t0)
269  syscall
270
271  beq $t1, $zero, run_completed # se le esecuzioni rimanenti sono zero, stampa "task
terminato"
272                                # altrimenti
273  li $v0, 4 # stampa "task eseguito"
274  la $a0, executed_msg
275  syscall
276
277  move $a0, $t0 # prepara il primo parametro: puntatore al task eseguito
278  move $a1, $t8 # secondo: puntatore d'inizio della lista A
279  move $a2, $t9 # terzo: puntatore d'inizio della lista B
280
281  jal insert # chiama la procedura insert, per reinserire il task eseguito al posto
giusto
282
283  move $t8, $v0 # recupera come valori di ritorno il puntatore d'inizio A
284  move $t9, $v1 # ed il puntatore d'inizio B
285
286  j run_done # salta alla fine dell'esecuzione
287
288  run_completed: # se il task è terminato (0 esecuzioni rimanenti)
289  li $v0, 4 # stampa "task terminato"
290  la $a0, completed_msg
291  syscall
292                                # e non lo reinserisce nelle liste (eliminazione logica
definitiva)
293  run_done: # al termine dell'esecuzione
294  lw $ra, 8($sp) # recupera l'indirizzo di ritorno dallo stack
295  addi $sp, $sp, 12 # dealloca lo stack frame
296
297  move $v0, $t8 # imposta come valori di ritorno il puntatore d'inizio A

```



```

298     move $v1, $t9 # ed il puntatore d'inizio B
299     jr $ra        # torna al chiamante
300 ### run end ###
301
302 ### find_id ###
303 find_id:          # procedura per recuperare un task con un ID specifico
304     move $t0, $a0 # primo argomento: ID del task da recuperare
305     move $t8, $a1 # secondo: puntatore d'inizio della lista A
306
307     move $t1, $zero # puntatore al task con ID cercato, inizializzato come nullo
308
309 find_id_loop:
310     lw $t2, 8($t8) # carica l'ID del task attuale
311     beq $t2, $t0, find_id_found # se gli ID coincidono, allora il task è stato trovato
312     lw $t2, 28($t8) # altrimenti, carica il puntatore al task successivo
313     beq $t2, $zero, find_id_end # se è nullo, allora si è raggiunta la fine senza trovare
    il task
314     move $t8, $t2 # altrimenti passa al successivo
315     j find_id_loop # ed esegue un altro ciclo
316
317 find_id_found:    # se il task è stato trovato
318     move $t1, $t8 # salva il puntatore al task
319
320 find_id_end:      # alla fine della procedura
321     move $v0, $t1 # restituisce il puntatore al task trovato, o null se non è stato
    trovato
322     jr $ra        # torna al chiamante
323 ### find_id end ###
324
325 ### insert_task ###
326 insert_task:      # procedura per l'inserimento di un nuovo task
327                  # primo argomento: ignorato
328     move $s6, $a1 # secondo: puntatore d'inizio A
329     move $s7, $a2 # terzo: puntatore d'inizio B
330
331     addi $sp, $sp, -20 # alloca 20 byte (5 word) nello stack frame
332     sw $ra, 16($sp)   # salva nello stack: il puntatore di ritorno al chiamante
333     sw $s6, 12($sp)   # il puntatore d'inizio della lista A
334     sw $s7, 8($sp)    # ed il puntatore d'inizio della lista B
335
336     li $v0, 4          # stampa la stringa d'inizio dell'inserimento di un task
337     la $a0, insert_task_msg
338     syscall
339
340     li $v0, 9          # chiamata sbrk
341     li $a0, 36          # 36 byte totali (9 word)
342     syscall            # puntatore al precedente A (4 byte, 0-4)
343                        # puntatore al precedente B (4 byte, 4-8)
344                        # ID (4 byte, 8-12)
345                        # nome (8 byte, 12-20)
346                        # priorità (4 byte, 20-24)
347                        # esecuzioni rimanenti (4 byte, 24-28)
348                        # puntatore al successivo A (4 byte, 28-32)
349                        # puntatore al successivo B (4 byte, 32-36)
350
351     move $s0, $v0      # salva l'indirizzo d'inizio dei 36 byte allocati in $s0
352     sw $s0, 4($sp)     # e lo salva nello stack
353

```

```

354     sw $zero, 0($s0)    # salva nell'heap: il puntatore al precedente A
355     sw $zero, 4($s0)    # e il puntatore al precedente B
356
357 insert_task_id:
358     li $v0, 4           # chiede di inserire l'ID del task
359     la $a0, insert_id_msg
360     syscall
361
362     li $v0, 5           # legge un intero da input
363     syscall
364     move $s1, $v0
365
366     beq $s6, $zero, insert_task_id_store    # se la coda è vuota, salva l'ID inserito
367                                             #
368 altrimenti
369     sw $s1, 0($sp)      # salva l'ID inserito nello stack
370
371     move $a0, $s1       # prepara il primo argomento: ID inserito
372     move $a1, $s6       # secondo argomento: puntatore d'inizio A
373
374     jal find_id         # invoca la procedura find_id
375
376     move $t1, $v0       # salva in $t1 il risultato di find_id
377     lw $s6, 12($sp)     # recupera dallo stack: il puntatore d'inizio A
378     lw $s7, 8($sp)      # il puntatore d'inizio B
379     lw $s0, 4($sp)      # il puntatore al record allocato nell'heap
380     lw $t0, 0($sp)      # ed il valore dell'ID inserito
381
382     bne $t1, $zero, insert_task_id_duplicate # se il puntatore ritornato da find_id è non
nulla, esiste già un task con l'ID inserito
383
384     j insert_task_id_store    # altrimenti, può
memorizzare l'ID tranquillamente
385
386 insert_task_id_duplicate:    # se è già presente in coda un task con l'ID inserito
387     li $v0, 4               # stampa un messaggio d'errore
388     la $a0, task_msg
389     syscall
390
391     li $v0, 1
392     move $a0, $t0
393     syscall
394
395     li $v0, 4
396     la $a0, id_duplicate_msg
397     syscall
398
399     j insert_task_id        # torna nuovamente all'inserimento dell'ID
400
401 insert_task_id_store: # se non si sono trovati ID duplicati
402     sw $t0, 8($s0)     # memorizza l'ID nell'heap
403
404 insert_task_name:
405     li $v0, 4           # chiede di inserire il nome del task
406     la $a0, insert_name_msg
407     syscall
408
409     li $v0, 8           # legge una stringa
410     la $a0, buffer

```

```

409     li $a1, 1024
410     syscall
411
412     li $t0, 0 # contatore inizializzato a zero (caratteri copiati nell'heap)
413
414 insert_task_name_loop:
415     li $t1, 8 # carica l'intero 8
416     beq $t0, $t1, insert_task_prio # se il contatore ha raggiunto 8, salta all'
inserimento della priorità
417                                     # altrimenti
418     la $t1, buffer # carica l'indirizzo di base della stringa inserita
419     add $t1, $t1, $t0 # shifta l'indirizzo in avanti del numero di caratteri già
copiati (diciamo n)
420     lb $t2, 0($t1) # carica il carattere n-esimo della stringa
421
422     li $t3, 32 # carica l'intero
32 (primo carattere non speciale nella codifica ASCII)
423     blt $t2, $t3, insert_task_name_spaces # se il carattere letto è un carattere
speciale, inizia il loop per inserire spazi
424                                     #
altrimenti
425     addi $t1, $s0, 12 # carica l'indirizzo base del campo nome del task (indirizzo base
del record + 12 byte)
426     add $t1, $t1, $t0 # ci somma il numero di caratteri letti (prima posizione del nome
non copiata)
427     sb $t2, 0($t1) # copia il carattere nell'heap
428
429     addi $t0, $t0, 1 # incrementa il contatore
430     j insert_task_name_loop # esegue un altro ciclo
431
432 insert_task_name_spaces: # se si è trovato un carattere speciale
433     li $t1, 8 # se si sono copiati 8 caratteri
434     beq $t0, $t1, insert_task_prio # finisce il ciclo
435                                     # altrimenti
436     addi $t1, $s0, 12 # carica l'indirizzo della prima posizione del nome disponibile
437     add $t1, $t1, $t0
438     li $t2, ' ' # e ci copia uno spazio
439     sb $t2, 0($t1)
440
441     addi $t0, $t0, 1 # incrementa il contatore
442     j insert_task_name_spaces # ed esegue un altro ciclo
443
444 insert_task_prio:
445     li $v0, 4 # chiede di inserire la priorità del task
446     la $a0, insert_prio_msg
447     syscall
448
449     li $v0, 5 # legge un intero da input
450     syscall
451
452     li $t0, 0 # se si è inserita una priorità minore di 0
453     blt $v0, $t0, insert_task_prio
454     li $t0, 9 # o se si è inserita una priorità maggiore di 9
455     bgt $v0, $t0, insert_task_prio # chiede nuovamente la priorità del task
456     sw $v0, 20($s0) # altrimenti salva nell'heap
457
458 insert_task_cycles:
459     li $v0, 4 # chiede di inserire i cicli di esecuzione

```

```

totali del task
460  la $a0, insert_cycles_msg
461  syscall
462
463  li $v0, 5 # legge un intero da input
464  syscall
465
466  li $t0, 1 # se si sono inseriti meno di 1 ciclo
467  blt $v0, $t0, insert_task_cycles
468  li $t0, 99 # o se si sono inseriti più di 99 cicli
469  bgt $v0, $t0, insert_task_cycles # chiede nuovamente i cicli del task
470  sw $v0, 24($s0) # altrimenti salva nell'heap
471
472  sw $zero, 28($s0) # salva nell'heap: il puntatore al successivo A
473  sw $zero, 32($s0) # ed il puntatore al successivo B
474
475  move $a0, $s0 # prepara il primo argomento: puntatore al task appena creato (da
inserire)
476  move $a1, $s6 # secondo: puntatore d'inizio A
477  move $a2, $s7 # terzo: puntatore d'inizio B
478
479  jal insert # invoca la procedura insert
480
481  move $s6, $v0 # recupera i valori di ritorno di insert: puntatore d'inizio A
482  move $s7, $v1 # puntatore d'inizio B
483
484  li $v0, 4 # stampa il corretto inserimento del
task
485  la $a0, insert_task_done_msg
486  syscall
487
488  lw $ra, 16($sp) # recupera l'indirizzo di ritorno al chiamante
489  addi $sp, $sp, 20 # dealloca lo stack
490
491  move $v0, $s6 # prepara i valori di ritorno: puntatore alla lista A
492  move $v1, $s7 # puntatore alla lista B
493
494  jr $ra # torna al chiamante
495  ### insert_task end ###
496
497  ### run_first ###
498  run_first: # procedura per eseguire il task in testa alla coda
499  move $t7, $a0 # primo argomento: politica di scheduling
500  move $t8, $a1 # secondo: puntatore d'inizio A
501  move $t9, $a2 # terzo: puntatore d'inizio B
502
503  li $v0, 4 # stampa la stringa d'inizio dell'esecuzione del primo task
504  la $a0, run_first_msg
505  syscall
506
507  li $t0, 'a' # carica il carattere 'a'
508  bne $t7, $t0, run_first_cycles # se la politica di scheduling non è A, passa all'
inizializzazione per la politica B
509  move $t1, $t8 # altrimenti, carica il puntatore d'
inizio di A
510
511  j run_first_check_empty # controlla se la coda è vuota
512

```

```

513 run_first_cycles:      # se la politica è B
514     move $t1, $t9      # carica il puntatore d'inizio B
515
516 run_first_check_empty:      # controlla se la coda è vuota
517     beq $t1, $zero, run_first_empty # se il puntatore d'inizio è nullo, la coda è vuota
518                                     # altrimenti
519 run_first_loop:
520     bne $t7, $t0, run_first_load_next_cycles # se la politica non è A, carica il seguente
521     della lista B
522                                     #
523     altrimenti
524     lw $t2, 28($t1)      # carica il seguente della lista A
525     j run_first_check_next # salta al controllo dell'elemento successivo
526
527 run_first_load_next_cycles: # se la politica non era A
528     lw $t2, 32($t1)      # carica il successivo della lista B
529
530 run_first_check_next:      # controllo dell'elemento successivo
531     beq $t2, $zero, run_first_found # se il puntatore all'elemento seguente è nullo,
532     allora il task in testa è stato trovato
533                                     # altrimenti
534     move $t1, $t2        # passa al seguente
535     j run_first_loop     # ed esegue un altro ciclo
536
537 run_first_found:          # quando l'elemento in testa è trovato
538     addi $sp, $sp, -4 # alloca 4 byte nello stack frame (1 word)
539     sw $ra, 0($sp)      # salva l'indirizzo di ritorno nello stack
540
541     move $a0, $t1        # prepara il primo parametro: puntatore al task in testa alla coda
542     move $a1, $t8        # secondo: puntatore d'inizio A
543     move $a2, $t9        # terzo: puntatore d'inizio B
544
545     jal run # invoca la procedura run
546
547     move $t8, $v0        # recupera i valori di ritorno di run: puntatore d'inizio A
548     move $t9, $v1        # e puntatore d'inizio B
549
550     lw $ra, 0($sp)      # recupera l'indirizzo di ritorno al chiamante dallo stack
551     addi $sp, $sp, 4    # e dealloca lo stack frame
552
553     j run_first_done # salta alla terminazione (con successo) della procedura
554     run_first
555
556 run_first_empty:          # se la coda era vuota
557     li $v0, 4           # stampa un messaggio indicando l'errore
558     la $a0, empty_msg
559     syscall
560
561     li $v0, 4
562     la $a0, run_not_done_msg
563     syscall
564
565     j run_first_return   # salta alle istruzioni di ritorno
566
567 run_first_done:          # se la procedura è terminata con successo
568     li $v0, 4           # stampa un messaggio di terminazione
569     la $a0, run_done_msg
570     syscall

```

```

567
568 run_first_return:
569     move $v0, $t8    # prepara i valori di ritorno: puntatore d'inizio A
570     move $v1, $t9    # e puntatore d'inizio B
571
572     jr $ra           # torna al chiamante
573 ### run_first end ###
574
575 ### run_id ###
576 run_id:                                     # procedura per eseguire un task specifico
577     addi $sp, $sp, -16    # alloca 16 byte nello stack frame (4 word)
578     sw $ra, 12($sp)      # salva l'indirizzo di ritorno nello stack
579
580                                     # primo argomento: ignorato
581     move $s6, $a1        # secondo: puntatore d'inizio A
582     move $s7, $a2        # terzo: puntatore d'inizio B
583
584     sw $s6, 8($sp)       # salva nello stack: il puntatore d'inizio A
585     sw $s7, 4($sp)       # ed il puntatore d'inizio B
586
587     li $v0, 4            # stampa la stringa d'inizio dell'esecuzione di un task
588     la $a0, run_id_msg   specifico
589     syscall
590
591     beq $s6, $zero, run_id_empty    # se il puntatore d'inizio è nullo, allora la coda è
592     vuota
593
594 run_id_choose:
595     li $v0, 4            # chiede di inserire l'ID del task da eseguire
596     la $a0, choose_id_msg
597     syscall
598
599     li $v0, 4
600     la $a0, choose_id_run_msg
601     syscall
602
603     li $v0, 5            # legge un intero da input
604     syscall
605     move $s0, $v0        # lo salva in $s0
606     sw $s0, 0($sp)       # e lo salva nello stack
607
608     move $a0, $s0        # prepara il primo parametro d'invocazione: ID inserito
609     move $a1, $s6        # secondo: puntatore d'inizio A
610
611     jal find_id          # invoca la procedura find_id
612
613     move $t0, $v0        # recupera il valore di ritorno di find_id
614     lw $s6, 8($sp)       # recupera dallo stack: il puntatore d'inizio A
615     lw $s7, 4($sp)       # ed il puntatore d'inizio B
616
617     beq $t0, $zero, run_id_not_found # se il puntatore restituito da find_id è nullo,
618     allora non c'è nessun task con l'ID inserito
619                                     # altrimenti
620     move $a0, $t0        # prepara il primo parametro: puntatore al task con ID selezionato
621     move $a1, $s6        # secondo: puntatore d'inizio A
622     move $a2, $s7        # terzo: puntatore d'inizio B

```

```

622     jal run    # invoca la procedura run
623
624     move $s6, $v0 # recupera i valori di ritorno: puntatore d'inizio A
625     move $s7, $v1 # e puntatore d'inizio B
626
627     j run_id_done # salta alla terminazione con successo della procedura
628
629 run_id_not_found:    # se il task con ID selezionato non è stato trovato
630     li $v0, 4        # stampa un messaggio d'errore
631     la $a0, task_msg
632     syscall
633
634     li $v0, 1
635     lw $a0, 0($sp)
636     syscall
637
638     li $v0, 4
639     la $a0, id_not_found_msg
640     syscall
641
642     j run_id_choose # e torna all'inserimento dell'ID
643
644 run_id_empty:        # se la coda era vuota
645     li $v0, 4        # stampa un messaggio d'errore
646     la $a0, empty_msg
647     syscall
648
649     li $v0, 4
650     la $a0, run_not_done_msg
651     syscall
652
653     j run_id_return # e ritorna
654
655 run_id_done:         # se l'esecuzione è eseguita con successo
656     li $v0, 4        # stampa un messaggio di corretta terminazione
657     la $a0, run_done_msg
658     syscall
659
660 run_id_return:
661     move $v0, $s6 # prepara i valori di ritorno: puntatore d'inizio A
662     move $v1, $s7 # e puntatore d'inizio B
663
664     lw $ra, 12($sp) # recupera l'indirizzo di ritorno dallo stack
665     addi $sp, $sp, 16 # e dealloca lo stack frame
666
667     jr $ra # torna al chiamante
668 ### run_id end ###
669
670 ### delete_id ###
671 delete_id:          # analogo a run_id, cambia soltanto la parte centrale
672     addi $sp, $sp, -16
673     sw $ra, 12($sp)
674
675     move $s6, $a1
676     move $s7, $a2
677
678     sw $s6, 8($sp)
679     sw $s7, 4($sp)

```

```

680
681     li $v0, 4
682     la $a0, delete_id_msg
683     syscall
684
685     beq $s6, $zero, delete_id_empty
686
687 delete_id_choose:
688     li $v0, 4
689     la $a0, choose_id_msg
690     syscall
691
692     li $v0, 4
693     la $a0, choose_id_delete_msg
694     syscall
695
696     li $v0, 5
697     syscall
698     move $s0, $v0
699     sw $s0, 0($sp)
700
701     move $a0, $s0
702     move $a1, $s6
703
704     jal find_id
705
706     move $t0, $v0
707     lw $s6, 8($sp)
708     lw $s7, 4($sp)
709
710     beq $t0, $zero, delete_id_not_found
711
712     move $a0, $t0
713     move $a1, $s6
714     move $a2, $s7
715
716     jal detach # invoca la procedura detach
717
718     move $s6, $v0
719     move $s7, $v1
720
721     j delete_id_done
722
723 delete_id_not_found:
724     li $v0, 4
725     la $a0, task_msg
726     syscall
727
728     li $v0, 1
729     lw $a0, 0($sp)
730     syscall
731
732     li $v0, 4
733     la $a0, id_not_found_msg
734     syscall
735
736     j delete_id_choose
737

```



```

738 delete_id_empty:
739     li $v0, 4
740     la $a0, empty_msg
741     syscall
742
743     li $v0, 4
744     la $a0, delete_id_not_done_msg
745     syscall
746
747     j delete_id_return
748
749 delete_id_done:
750     li $v0, 4
751     la $a0, delete_id_done_msg
752     syscall
753
754 delete_id_return:
755     move $v0, $t8
756     move $v1, $t9
757
758     lw $ra, 12($sp)
759     addi $sp, $sp, 16
760
761     jr $ra
762 ### delete_id end ###
763
764 ### change_prio ###
765 change_prio:          # analogo a run_id, cambia soltanto la parte centrale
766     addi $sp, $sp, -20
767     sw $ra, 16($sp)
768
769     move $s6, $a1
770     move $s7, $a2
771
772     sw $s6, 12($sp)
773     sw $s7, 8($sp)
774
775     li $v0, 4
776     la $a0, change_prio_msg
777     syscall
778
779     beq $s6, $zero, change_prio_empty
780
781 change_prio_choose:
782     li $v0, 4
783     la $a0, choose_id_msg
784     syscall
785
786     li $v0, 4
787     la $a0, choose_id_change_prio_msg
788     syscall
789
790     li $v0, 5
791     syscall
792     move $s0, $v0
793     sw $s0, 4($sp)
794
795     move $a0, $s0

```

```

796     move $a1, $s6
797
798     jal find_id
799
800     move $s1, $v0
801     sw $s1, 0($sp)
802     lw $s6, 12($sp)
803     lw $s7, 8($sp)
804
805     beq $s1, $zero, change_prio_not_found
806
807 change_prio_insert_new:
808     li $v0, 4          # chiede la nuova priorità del task
809     la $a0, new_prio_msg
810     syscall
811
812     li $v0, 5          # legge un intero
813     syscall
814     move $t0, $v0      # e lo salva in $t0
815
816     li $t1, 0          # se si è inserita una priorità minore di 0
817     blt $t0, $t1, change_prio_insert_new
818     li $t1, 9          # o se si è inserita una priorità maggiore di 9
819     bgt $t0, $t1, change_prio_insert_new # chiedi nuovamente la priorità del task
820     sw $t0, 20($s1)    # altrimenti aggiorna la priorità nell'heap
821
822     move $a0, $s1      # prepara il primo parametro: puntatore al task da modificare
823     move $a1, $s6      # secondo: puntatore d'inizio A
824     move $a2, $s7      # terzo: puntatore d'inizio B
825
826     jal detach         # invoca la procedura detach
827
828     lw $s1, 0($sp)     # recupera il puntatore al task da modificare dallo stack
829     move $s6, $v0      # recupera i valori di ritorno: puntatore d'inizio A
830     move $s7, $v1      # e puntatore d'inizio B
831
832     move $a0, $s1      # prepara i parametri (come prima)
833     move $a1, $s6
834     move $a2, $s7
835
836     jal insert         # invoca la procedura insert
837
838     move $s6, $v0      # recupera i valori di ritorno (come prima)
839     move $s7, $v1
840
841     j change_prio_done
842
843 change_prio_not_found:
844     li $v0, 4
845     la $a0, task_msg
846     syscall
847
848     li $v0, 1
849     lw $a0, 4($sp)
850     syscall
851
852     li $v0, 4
853     la $a0, id_not_found_msg

```

```

854     syscall
855
856     j change_prio_choose
857
858 change_prio_empty:
859     li $v0, 4
860     la $a0, empty_msg
861     syscall
862
863     li $v0, 4
864     la $a0, change_prio_not_done_msg
865     syscall
866
867     j change_prio_return
868
869 change_prio_done:
870     li $v0, 4
871     la $a0, change_prio_done_msg
872     syscall
873
874 change_prio_return:
875     move $v0, $s6
876     move $v1, $s7
877
878     lw $ra, 16($sp)
879     addi $sp, $sp, 20
880
881     jr $ra
882 ### change_prio end ###
883
884 ### change_sched ###
885 change_sched: # procedura per cambiare la politica di scheduling
886     move $t7, $a0 # primo parametro: politica di scheduling attuale
887                 # secondo e terzo parametro: ignorati
888
889     li $v0, 4          # stampa la stringa d'inizio del cambio della politica di
890     la $a0, change_sched_msg
891     syscall
892
893     li $v0, 4          # stampa "Politica di scheduling attuale:"
894     la $a0, change_sched_old_msg
895     syscall
896
897     li $t0, 'a'        # carica il carattere 'a'
898     bne $t7, $t0, change_sched_to_prio # se la politica di scheduling attuale non è A,
899                                     # altrimenti
900     li $v0, 4          # stampa la politica attuale (su PRIORITÀ)
901     la $a0, sched_prio_msg
902     syscall
903
904     li $t7, 'b' # cambia la politica a B
905
906     li $v0, 4          # stampa la nuova politica (su ESECUZIONI RIMANENTI)
907     la $a0, change_sched_new_msg
908     syscall
909

```

```

910     li $v0, 4
911     la $a0, sched_cycles_msg
912     syscall
913
914     j change_sched_end # salta alla fine della procedura
915
916 change_sched_to_prio:      # se la politica non era A
917     li $v0, 4              # stampa la politica attuale (su ESECUZIONI RIMANENTI)
918     la $a0, sched_cycles_msg
919     syscall
920
921     li $t7, 'a' # cambia la politica ad A
922
923     li $v0, 4              # stampa la nuova politica (su PRIORITA)
924     la $a0, change_sched_new_msg
925     syscall
926
927     li $v0, 4
928     la $a0, sched_prio_msg
929     syscall
930
931 change_sched_end:
932     li $v0, 4              # stampa un messaggio di corretta terminazione della
933     la $a0, change_sched_done_msg
934     syscall
935
936     move $v0, $t7 # imposta la nuova politica di scheduling come valore di ritorno
937
938     jr $ra # ritorna al chiamante
939 ### change_sched end ###
940
941 ### Main ###
942 main:
943     li $s5, 'a' # inizializza la politica di scheduling ad A (scheduling su priorit
944     move $s6, $zero # ed il puntatore alla coda A
945     move $s7, $zero # e alla coda B a 0 (nessun elemento in coda)
946
947 command_selection:
948     li $v0, 4              # stampa la stringa per la selezione del comando
949     la $a0, insert_command
950     syscall
951
952     li $v0, 5              # legge un intero in input
953     syscall
954     move $t0, $v0 # salva l'intero letto in $t0
955
956     li $v0, 4              # stampa una newline
957     la $a0, newline
958     syscall
959
960     move $a0, $s5 # prepara il primo parametro (politica di scheduling)
961     move $a1, $s6 # il secondo (puntatore ad A)
962     move $a2, $s7 # ed il terzo (puntatore a B)
963
964     li $t1, 1              # se l'intero letto è 1
965     beq $t0, $t1, jump_insert_task # il comando da eseguire è l'inserimento di un nuovo

```

```

task
966     li $t1, 2                # se è 2
967     beq $t0, $t1, jump_run_first # il comando da eseguire è l'esecuzione del primo task
968     li $t1, 3                # se è 3
969     beq $t0, $t1, jump_run_id  # il comando da eseguire è l'esecuzione di un task
specifico
970     li $t1, 4                # se è 4
971     beq $t0, $t1, jump_delete_id # il comando da eseguire è l'eliminazione di un task
specifico
972     li $t1, 5                # se è 5
973     beq $t0, $t1, jump_change_prio # il comando da eseguire è la modifica della priorit
à di un task specifico
974     li $t1, 6                # se è 6
975     beq $t0, $t1, jump_change_sched # il comando da eseguire è il cambio della politica
di scheduling
976     li $t1, 7                # se è 7
977     beq $t0, $t1, exit        # il comando da eseguire è la terminazione del
programma
978                                #altrimenti
979     li $v0, 4                # stampa il menu
980     la $a0, menu
981     syscall
982
983     j command_selection # e torna alla selezione del comando
984
985 jump_insert_task:            # prepara il necessario per il salto al comando d'inserimento
986     addi $sp, $sp, -4        # alloca 4 byte nello stack frame
987     sb $s5, 0($sp)          # salva la politica di scheduling nello stack
988
989     jal insert_task # salta alla procedura per l'inserimento di un task
990                                # recupera i risultati della procedura
991     move $s6, $v0 # ovvero il puntatore ad A
992     move $s7, $v1 # ed il puntatore a B
993
994     lb $s5, 0($sp)          # recupera la politica di scheduling dallo stack
995     addi $sp, $sp, 4        # dealloca i 4 byte dallo stack
996
997     j print_queue # salta alle istruzioni per la stampa della coda
998
999 jump_run_first:            # jump_run_first, jump_run_id, jump_delete_id e jump_change_prio
analoghi a jump_insert_task
1000     addi $sp, $sp, -4
1001     sb $s5, 0($sp)
1002
1003     jal run_first
1004
1005     move $s6, $v0
1006     move $s7, $v1
1007
1008     lb $s5, 0($sp)
1009     addi $sp, $sp, 4
1010
1011     j print_queue
1012
1013 jump_run_id:
1014     addi $sp, $sp, -4
1015     sb $s5, 0($sp)
1016

```

```

1017     jal run_id
1018
1019     move $s6, $v0
1020     move $s7, $v1
1021
1022     lb $s5, 0($sp)
1023     addi $sp, $sp, 4
1024
1025     j print_queue
1026
1027 jump_delete_id:
1028     addi $sp, $sp, -4
1029     sb $s5, 0($sp)
1030
1031     jal delete_id
1032
1033     move $s6, $v0
1034     move $s7, $v1
1035
1036     lb $s5, 0($sp)
1037     addi $sp, $sp, 4
1038
1039     j print_queue
1040
1041 jump_change_prio:
1042     addi $sp, $sp, -4
1043     sb $s5, 0($sp)
1044
1045     jal change_prio
1046
1047     move $s6, $v0
1048     move $s7, $v1
1049
1050     lb $s5, 0($sp)
1051     addi $sp, $sp, 4
1052
1053     j print_queue
1054
1055 jump_change_sched:    # analogo agli altri jump
1056     addi $sp, $sp, -8    # con la differenza che nello stack si salvano i due puntatori
1057     sw $s6, 4($sp)      # e la procedura ritorna la nuova politica di scheduling
1058     sw $s7, 0($sp)
1059
1060     jal change_sched
1061
1062     move $s5, $v0
1063
1064     lw $s6, 4($sp)
1065     lw $s7, 0($sp)
1066     addi $sp, $sp, 8
1067
1068     j print_queue
1069
1070 print_queue:          # istruzioni per la stampa della coda
1071     li $v0, 4          # stampa "Scheduling su"
1072     la $a0, scheduling_msg # il nome effettivo viene poi stampato a seconda dello
                           scheduling attuale
1073     syscall

```

```

1074     li $t0, 'a' # carica il carattere 'a', per poterlo
1075     comparare con la politica attuale
1076     bne $s5, $t0, print_queue_cycles_begin # se la politica non è A, esegui l'
1077     # altrimenti
1078     move $t1, $s6 # inizializza il puntatore $t1 col puntatore di A
1079
1080     li $v0, 4 # stampa "PRIORITÀ"
1081     la $a0, sched_prio_msg
1082     syscall
1083
1084     j print_queue_header # salta alla stampa dell'header della tabella
1085
1086 print_queue_cycles_begin: # se la politica è B
1087     move $t1, $s7 # inizializza il puntatore $t1 col puntatore di B
1088
1089     li $v0, 4 # stampa "ESECUZIONI RIMANENTI"
1090     la $a0, sched_cycles_msg
1091     syscall
1092
1093 print_queue_header:
1094
1095     li $v0, 4 # stampa un delimitatore
1096     la $a0, print_delim
1097     syscall
1098
1099     li $v0, 4 # stampa l'header
1100     la $a0, print_header
1101     syscall
1102
1103     li $v0, 4 # stampa un delimitatore
1104     la $a0, print_delim
1105     syscall
1106
1107     beq $t1, $zero, print_queue_empty # se il puntatore d'inizio è zero, salta alla
1108     stampa della coda vuota
1109
1109 print_queue_loop:
1110     li $v0, 4 # stampa l'inizio di una riga
1111     la $a0, print_row_start
1112     syscall
1113
1114     lw $t2, 8($t1) # carica l'ID del task attuale
1115     li $t3, 9 # se è composto da due cifre
1116     bgt $t2, $t3, print_queue_id # stampa subito l'ID
1117     # altrimenti
1118     li $v0, 11 # stampa prima uno spazio extra
1119     li $a0, ' '
1120     syscall
1121
1122 print_queue_id:
1123     li $v0, 11 # stampa uno spazio
1124     li $a0, ' '
1125     syscall
1126
1127     li $v0, 1 # stampa l'ID
1128     move $a0, $t2

```

```

1129 syscall
1130
1131 li $v0, 11 # stampa uno spazio
1132 li $a0, ' '
1133 syscall
1134
1135 li $v0, 11 # ed un separatore
1136 li $a0, '/'
1137 syscall
1138
1139 li $v0, 4 # stampa cinque spazi
1140 la $a0, print_row_five_spaces
1141 syscall
1142
1143 li $v0, 1 # stampa la priorità
1144 lw $a0, 20($t1)
1145 syscall
1146
1147 li $v0, 4 # stampa altri cinque spazi
1148 la $a0, print_row_five_spaces
1149 syscall
1150
1151 li $v0, 11 # stampa un separatore
1152 li $a0, '/'
1153 syscall
1154
1155 li $v0, 11 # ed uno spazio
1156 li $a0, ' '
1157 syscall
1158
1159 li $t2, 12 # carica 12 (offset d'inizio del nome)
1160 li $t3, 19 # e 19 (offset di fine)
1161
1162 print_queue_name_loop:
1163 add $t4, $t2, $t1 # puntatore al carattere attuale (indirizzo base $t1 + offset
attuale $t2)
1164
1165 li $v0, 11 # stampa il carattere
1166 lb $a0, 0($t4)
1167 syscall
1168
1169 addi $t2, $t2, 1 # incrementa l'offset
1170 ble $t2, $t3, print_queue_name_loop # se l'offset è minore o uguale dell'offset
massimo, esegue un altro ciclo
1171
1172 li $v0, 11 # stampa due spazi
1173 li $a0, ' '
1174 syscall
1175 syscall
1176
1177 li $v0, 11 # ed un separatore
1178 li $a0, '/'
1179 syscall
1180
1181 li $v0, 4 # stampa otto spazi
1182 la $a0, print_row_eight_spaces
1183 syscall
1184

```



```

1185 lw $t2, 24($t1)           # carica il numero di cicli rimanenti
1186 li $t3, 9                 # se è composto da due cifre
1187 bgt $t2, $t3, print_queue_cycles # stampa subito il numero di cicli
1188                             # altrimenti
1189 li $v0, 11 # stampa prima uno spazio extra
1190 li $a0, ' '
1191 syscall
1192
1193 print_queue_cycles:
1194 li $v0, 1 # stampa il numero di esecuzioni rimanenti
1195 move $a0, $t2
1196 syscall
1197
1198 li $v0, 4 # stampa otto spazi
1199 la $a0, print_row_eight_spaces
1200 syscall
1201
1202 li $v0, 11 # stampa uno spazio extra
1203 li $a0, ' '
1204 syscall
1205
1206 li $v0, 11 # stampa un separatore
1207 li $a0, '/'
1208 syscall
1209
1210 li $v0, 4 # va a capo
1211 la $a0, newline
1212 syscall
1213
1214 li $v0, 4 # stampa un delimitatore
1215 la $a0, print_delim
1216 syscall
1217
1218 bne $s5, $t0, print_queue_load_next_cycles # controlla la politica di scheduling
1219 attuale
1220 lw $t1, 28($t1)           # se è A, carica il prossimo elemento della
1221 lista A
1222 j print_queue_check_next # e salta al controllo del prossimo
1223 elemento
1224 # altrimenti
1225 print_queue_load_next_cycles:
1226 lw $t1, 32($t1)           # carica il prossimo elemento della lista B
1227
1228 print_queue_check_next:
1229 bne $t1, $zero, print_queue_loop # se il prossimo elemento non è nullo, esegui un
1230 altro ciclo
1231 j print_end # altrimenti, salta alla fine della stampa
1232
1233 print_queue_empty:
1234 # in caso di coda vuota
1235 li $v0, 4 # stampa una riga dedicata
1236 la $a0, print_empty_queue
1237 syscall
1238
1239 li $v0, 4 # e stampa un delimitatore
1240 la $a0, print_delim
1241 syscall
1242
1243 print_end: # alla fine della coda

```

```

1239     li $v0, 4      # stampa una newline
1240     la $a0, newline
1241     syscall
1242
1243     j command_selection # e torna alla selezione del comando
1244
1245 exit :
1246     li $v0, 4      # stampa il messaggio di terminazione
1247     la $a0, exiting
1248     syscall
1249
1250     li $v0, 10 # uscita dal programma
1251     syscall
1252     ### Main end ###

```