



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)¹. 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){
```

¹ 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
4     pars = 0
    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. La stringa 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.

Infine, 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.

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.

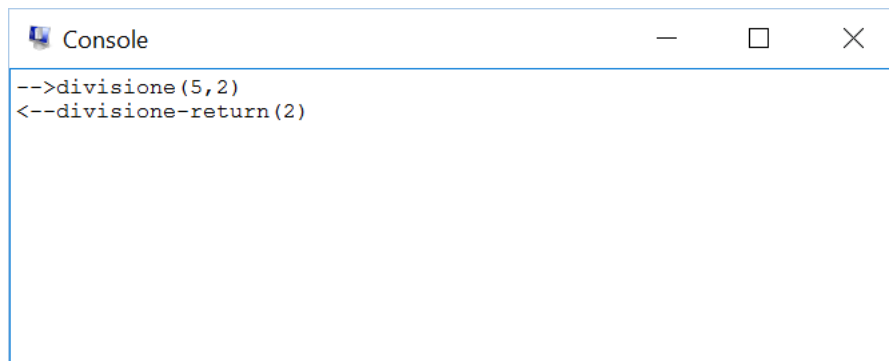
A screenshot of a console window titled "Console". The window has standard window controls (minimize, maximize, close) in the top right corner. The console displays two lines of text: "-->divisione(5,2)" on the first line and "<--divisione-return(2)" on the second line. The text is in a monospaced font.

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

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
-->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_
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 $t0, $a0 # inizializza $t0 con il puntatore d'inizio
116     move $t1, $zero # e $t1 con 0 ($t1 indica il numero di parentesi aperte ma non
117     ancora chiuse)
118
119 get_operands_search:
120     lb $t3, 0($t0)      # carica il carattere puntato da $t0
121     li $t4, '('
122     beq $t3, $t4, get_operands_open # controlla se è una parentesi aperta
123     li $t4, ')'
124     beq $t3, $t4, get_operands_close # una parentesi chiusa
125     li $t4, ','
126     beq $t3, $t4, get_operands_comma # oppure una virgola
127     j get_operands_advance # altrimenti va avanti senza fare niente
128
129 get_operands_open:
130     addi $t1, $t1, 1 # se si trova una parentesi aperta si incrementa $t1
131     j get_operands_advance # e si passa al prossimo carattere
132
133 get_operands_close:
134     addi $t1, $t1, -1 # se si trova una parentesi chiusa si decrementa $t1
135     j get_operands_advance # e si passa al prossimo carattere

```

```

135
136 get_operands_comma:
137     beq $t1, $zero, get_operands_found # se si trova una virgola e le parentesi sono
138     bilanciaste, 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 $t0, $t0, 1                  # incrementa di 1 il puntatore $t0
143     j get_operands_search             # e continua la ricerca
144
145 get_operands_found:
146     sw $t0, 4($sp)                   # salva il puntatore alla virgola nello stack
147
148     addi $t0, $t0, -1                 # decrementa il puntatore $t0 (carattere subito prima della
149     virgola)
150     move $a1, $t0                    # 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
177     sw $ra, 16($sp)                  # salva l'indirizzo di ritorno nello stack
178
179     move $t0, $a0                    # primo parametro: puntatore d'inizio
180     move $t1, $a1                    # secondo: puntatore di fine
181     move $t2, $a2                    # terzo: profondità della chiamata
182
183     sw $t0, 12($sp)                  # il puntatore d'inizio
184     sw $t1, 8($sp)                   # il puntatore di fine
185     sw $t2, 4($sp)                   # e la profondità della chiamata
186
187     move $a0, $t0                    # primo parametro: puntatore d'inizio
188     move $a1, $t1                    # secondo: puntatore di fine
189     move $a2, $t2                    # terzo: profondità della chiamata
190
191     jal print_call                   # invoca la procedura per la stampa dell'invocazione (con gli
192     stessi parametri)

```

```

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 $t2, $t0, $t1 # somma i due operandi ottenuti
204 sw $t2, 0($sp) # salva il risultato nello stack
205
206 la $a0, sum_return # carica l'indirizzo della stringa sum_return (primo parametro)
207 move $a1, $t2 # 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 $t0, $a0 # primo parametro: puntatore d'inizio
224 move $t1, $a1 # secondo: puntatore di fine
225 move $t2, $a2 # terzo: profondità della chiamata
226
227 sw $t0, 12($sp) # il puntatore d'inizio
228 sw $t1, 8($sp) # il puntatore di fine
229 sw $t2, 4($sp) # e la profondità della chiamata
230
231 move $a0, $t0 # primo parametro: puntatore d'inizio
232 move $a1, $t1 # secondo: puntatore di fine
233 move $a2, $t2 # 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 $t2, $t0, $t1 # sottrae il secondo operando al primo
253  sw $t2, 0($sp) # salva il risultato nello stack
254
255  la $a0, sub_return # carica l'indirizzo della stringa sub_return (primo parametro)
256  move $a1, $t2 # 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 $t0, $a0 # primo parametro: puntatore d'inizio
273  move $t1, $a1 # secondo: puntatore di fine
274  move $t2, $a2 # terzo: profondità della chiamata
275
276  sw $t0, 12($sp) # il puntatore d'inizio
277  sw $t1, 8($sp) # il puntatore di fine
278  sw $t2, 4($sp) # e la profondità della chiamata
279
280  move $a0, $t0 # primo parametro: puntatore d'inizio
281  move $a1, $t1 # secondo: puntatore di fine
282  move $a2, $t2 # 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 $t2, $t0, $t1 # moltiplica i due operandi ottenuti

```

```

302     sw $t2, 0($sp)      # salva il risultato nello stack
303
304     la $a0, prod_return # carica l'indirizzo della stringa prod_return (primo parametro)
305     move $a1, $t2       # 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 $t0, $a0       # primo parametro: puntatore d'inizio
322     move $t1, $a1       # secondo: puntatore di fine
323     move $t2, $a2       # terzo: profondità della chiamata
324
325     sw $t0, 12($sp)     # il puntatore d'inizio
326     sw $t1, 8($sp)      # il puntatore di fine
327     sw $t2, 4($sp)      # e la profondità della chiamata
328
329     move $a0, $t0       # primo parametro: puntatore d'inizio
330     move $a1, $t1       # secondo: puntatore di fine
331     move $a2, $t2       # 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 $t2, $t0, $t1   # divide (operazione quoziente) il primo operando per il secondo
351     sw $t2, 0($sp)      # salva il risultato nello stack
352
353     la $a0, div_return  # carica l'indirizzo della stringa div_return (primo parametro)
354     move $a1, $t2       # 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 (
                          # $v0)
413
414     beq $a0, $a1, analyze_end # se i puntatori d'inizio e fine coincidono allora era l'

```

```

ultimo carattere
                                # altrimenti
415
    li $t2, 10
416
    mul $v0, $v0, $t2    # moltiplica il valore fin'ora calcolato per 10
417
    addi $a0, $a0, 1    # incrementa il puntatore di 1 (prossimo carattere)
418
419
    j analyze_int_loop # ed esegue un altro ciclo
420
421
analyze_end:
    # nel caso delle chiamate a procedura, il valore di ritorno è lo stesso (
422
    # valutazione dell'espressione)
423
    lw $ra, 0($sp)    # recupera l'indirizzo di ritorno
424
    addi $sp, $sp, 4    # dealloca lo stack
425
    jr $ra            # torna al chiamante
426
### Analyze end ###
427
428
### Main ###
429
main:
430
    li $v0, 13    # apre il file (syscall)
431
    la $a0, file    # carica il nome del file
432
    li $a1, 0    # sola lettura
433
    li $a2, 0
434
    syscall
435
    move $t0, $v0    # salva il file descriptor
436
437
    li $v0, 14    # lettura (syscall)
438
    move $a0, $t0    # carica il file descriptor
439
    la $a1, buffer    # carica il buffer
440
    li $a2, 1024    # specifica la dimensione
441
    syscall
442
    move $t1, $v0    # salva la lunghezza della stringa
443
444
    li $v0, 16    # chiusura del file (syscall)
445
    move $a0, $t0    # carica il file descriptor
446
    syscall
447
448
    la $t2, buffer    # calcola il puntatore d'inizio
449
    addi $t2, $t2, 1    # saltando le " iniziali
450
    la $t3, buffer    # il puntatore di fine
451
    add $t3, $t3, $t1    # sommando la lunghezza della stringa
452
    addi $t3, -2    # e sottraendo 2 (evita le " finali)
453
    li $t4, 0    # e la profondità delle chiamate iniziale
454
455
    move $a0, $t2    # prepara i parametri
456
    move $a1, $t3
457
    move $a2, $t4
458
459
    jal analyze # inizia analizzando l'intera stringa
460
    # ignora il valore di ritorno
461
462
    li $v0, 10    # uscita dal programma (syscall)
463
    syscall
464
### Main end ###
465

```

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)
           case 4:
16          start_A, start_B = delete_id(start_A, start_B)
           case 5:
18          start_A, start_B = change_prio(start_A, start_B)
           case 6:
20          sched = change_sched(sched)
           case 7:
22          print("Terminazione del programma.")
              exit()
24          default:
              print("Menu: ...")
26      }

28      if(sched == 'a'){
```



```

        i = start_A
    } else {
        i = start_B
    }
    delim = "+---+---+---+---+---+---+"
    header = "| ID | PRIORITA' | NOME TASK | ESECUZ. RIMANENTI |"
    println(delim)
    println(header)
    println(delim)
    if(i == null){
        println("Coda vuota!")
        println(delim)
        continue
    }
    while(i != null){
        print("/ ")
        id = i.id
        if (id <= 9){
            print(" " + id)
        } else {
            print(" " + id)
        }
        print(" / " + i.prio + " / " + i.name + " / ")
        cycles = i.cycles
        if (cycles <= 9){
            print(" " + cycles)
        } else {
            print(cycles)
        }
        println(" /")
        println(delim)
        if(sched == 'a'){
            i = i.next_A
        } else {
            i = i.next_B
        }
    }
}
}
}

```

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
           break
32       }
   }
34   task.next_A = null
   task.next_B = null
36
   start_A, start_B = insert(task, start_A, start_B)
38
   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
   i = start_A
4   while(i != null){
       if(id == i.id){
6           task = i
           break
8       }
       i = i.next_A
10  }
   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
    }

12    while(true){
14        if(sched == 'a'){
            next = i.next_A
16        } else {
            next = i.next_B
18        }
        if(next == null){
20            break
        }
22        i = next
    }

24    start_A, start_B = run(i, start_A, start_B)
```

```

26     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
    start_A, start_B = detach(task, start_A, start_B)
4     if(task.cycles > 0){
        start_A, start_B = insert(task, start_A, start_B)
6     }
    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
32    task.next_A = null
34    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

```

```

8   while(true){
    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
    return start_A, start_B
20 }

```

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

Simulazioni

Codice MIPS