Elaborato Assembly

Corso di Architettura degli Elaboratori A.A. 2023/2024
Prof. Franco Fummi, Prof. Michele Lora

Tommi Bimbato VR500751, Antonio Iovine VR504083

19 giugno 2024

Indice

Sommario

L'obiettivo del progetto è sviluppare un software per la pianificazione delle attività di un sistema produttivo. La produzione è organizzata in slot temporali uniformi, durante i quali un solo prodotto può essere in fase di lavorazione. Il software consentirà di ottimizzare la pianificazione delle attività secondo due algoritmi di pianificazioni differenti. L'intero software è stato sviluppato in linguaggio Assembly (Sintassi AT&T) e testato su un insieme di dati di prova allegati a questa documentazione.

Capitolo 1

Introduzione

1.1 Approccio progettuale

L'elaborato è stato condotto seguendo un approccio metodologico strutturato. Inizialmente, è stata eseguita un'analisi dettagliata per identificare i requisiti e le funzionalità principali del software. Questo processo ha consentito una comprensione completa del contesto operativo e degli obiettivi da raggiungere.

Successivamente, è stata sviluppata una bozza del software utilizzando il linguaggio di programmazione C. Questo ha permesso la traduzione dei requisiti in una struttura logica e l'identificazione dell'architettura generale del software.

Parallelamente, sono stati definiti gli spazi di memoria necessari per la memorizzazione dei dati durante l'esecuzione del programma. Ciò ha garantito un utilizzo efficiente delle risorse disponibili.

Infine, sono stati eseguiti test approfonditi per verificare il corretto funzionamento del programma e identificare eventuali aree di miglioramento. L'iterazione su questo processo ha portato a modifiche e ottimizzazioni fino al raggiungimento di un livello soddisfacente di prestazioni e funzionalità.

1.2 Analisi delle specifiche

La produzione è organizzata in slot temporali uniformi, durante i quali un solo prodotto può essere in fase di lavorazione. Il programma analizza una serie di prodotti, ognuno caratterizzato da un identificativo, una durata, una scadenza e una priorità secondo le specifiche seguenti:

- Identificativo: un codice da 1 a 127;
- Durata: il numero di slot temporali per il completamento (da 1 a 10);
- Scadenza: il limite massimo di tempo entro cui il prodotto deve essere completato (da 1 a 100);
- Priorità: un valore da 1 a 5, che indica sia la priorità che la penalità per il ritardo sulla scadenza[1].

Al termine della pianificazione, il programma calcolerà la penale dovuta agli eventuali ritardi di produzione.

1.2.1 Input

L'utente invoca il programma "pianificatore" e fornisce due file come parametri da linea di comando, il primo viene considerato input, mentre il secondo viene utilizzato per salvare i risultati della pianificazione. Ad esempio:q

```
./Pianificatore Ordini.txt output.txt
```

[1] Il valore 5 indica la priorità più alta.

In questo caso, il programma caricherà gli ordini dal file `Ordini.txt` e salverà le statistiche stampate a video nel file `output.txt`. Se l'utente fornisce solo un parametro, il salvataggio della pianificazione su file verrà ignorato.

Il file degli ordini dovrà avere un prodotto per riga, con tutti i parametri separati da virgola. Ad esempio, se l'ordine fosse:

```
ID: 4; Durata: 10; Scadenza: 12; Priorità: 4;
```

Il file dovrà contenere la seguente riga:

```
4,10,12,4
```

1.2.2 Algoritmi di pianificazione

Una volta letto il file, il programma visualizzerà il menu principale, permettendo all'utente di selezionare l'algoritmo di pianificazione desiderato. Le opzioni disponibili sono:

1. Earliest Deadline First (EDF): Si pianificano per primi i prodotti con scadenza più vicina. In caso di parità nella scadenza, si considera la priorità più alta.
2. Highest Priority First (HPF): Si pianificano per primi i prodotti con la priorità più alta. In caso di parità di priorità, si considera la scadenza più vicina.

L'utente può selezionare uno dei due algoritmi per la pianificazione delle attività del sistema produttivo.

1.3 Sviluppo

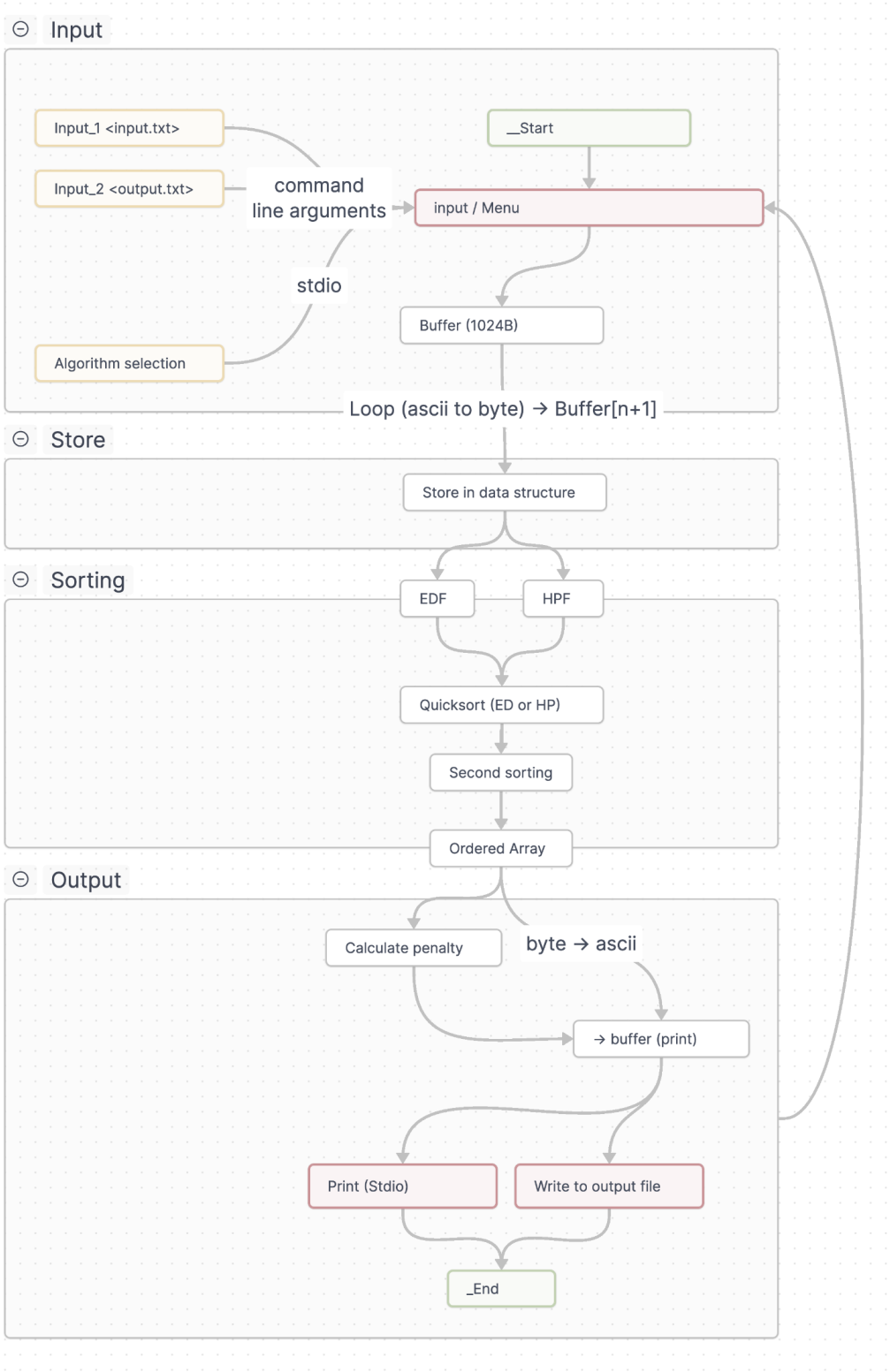

Figura 1.1: Schema base del funzionamento del software. [alcune funzionalità sono state omesse per leggibilità dello schema.]

1.3.1 Acquisizione e struttura dati

Il software acquisisce l'intero contenuto del file di input e lo memorizza in una variabile nel buffer, parallelamente ne calcola la lunghezza (numero di righe) per allocare la memoria necessaria per la struttura dati vera e propria. Successivamente, il programma procede a leggere il contenuto del buffer, converte le informazioni dal formato ASCII al formato numerico e le memorizza in un array di struct. La peculiarità di questa struttura dati è che lo spazio occupato da un prodotto (ID, durata, scadenza e priorità) è di 4 byte totali, pari alla dimensione di un registro interno del processore considerato per l'applicazione del software.

1.3.2 Ordinamento dei prodotti

Acquisita la struttura dati, il programma acquisisce la scelta dell'algoritmo di pianificazione da parte dell'utente e procede con l'ordinamento dei prodotti in base alla scelta effettuata. Il sorting è gestito da una variabile che definisce la priorità di ordinamento (HPF o EDF) che viene passata alla funzione di ordinamento come parametro. La funzione di ordinamento, a sua volta, si occupa di confrontare i prodotti in base alla priorità scelta e di riordinarli in base a tale criterio nella struttura dati stessa.

Per l'effettivo ordinamento dei prodotti, il software utilizza un algoritmo di sorting di tipo "insertion sort" che si è dimostrato il più efficiente per la dimensione dei dati in ingresso. Il secondo ordinamento, ovvero quello che compara il parametro "scadenza" in caso di parità di priorità o "priorità in caso di pari scadenza, è gestito da una funzione basata su algoritmo di bubble sort.

1.3.3 Output

Nel caso venisse specificato un file di output, il software stamperà su di esso i risultati della pianificazione secondo il seguente schema:

```
<Algoritmo di pianificazione scelto>:
<ID del prodotto>:<Inizio produzione>
[...]
Conclusione:<timeslot termine produzione>
Penalty:<penalità totale>
```

Esempio:

```
Pianificazione EDF:
1:0
4:2
…
9:55
Conclusione: 63
Penalty: 0
```

L'output della pianificazione viene comunque visualizzato a video.

1.3.4 Penalità

Se un prodotto non viene completato entro la scadenza, il programma calcola la penalità in base alla priorità del prodotto e alla quantità di slot temporali di ritardo. La penalità è calcolata come segue:

$$\text{Penalità} = \text{Priorità} \times \text{Ritardo}$$

1.4 Codice

1.4.1 Struttra della cartella di progetto

```
VR500751_VR504083
├── bin
│   └── *Pianificatore*
├── obj
├── Ordini
│   ├── EDF.txt
│   ├── NONE.txt
│   └── BOTH.txt
├── src
│   ├── AskInput.s
│   ├── AskOutput.s
│   ├── btoa.s
│   ├── BufferToArray.s
│   ├── EqualSort.s
│   ├── FinalData.s
│   ├── InsertionSort.s
│   ├── Intestazione.s
│   ├── menu.s
│   ├── main.s
│   └── textin.s
└── Makefile
```

1.4.2 File sorgenti

Il programma è composto dai seguenti file sorgente in linguaggio assembly:

- `AskInput.s`: funzione per richiedere il nome del file di input all'utente;
- `AskOutput.s`: funzione per richiedere il nome del file di output all'utente;
- `textin.s`: funzione per la lettura di una scelta da tastiera.
- `btoa.s`: funzione loop per la conversione di un byte in ASCII;
- `BufferToArray.s`: funzione per la conversione del buffer di lettura in un array di struct;
- `EqualSort.s`: funzione per il secondo ordinamento dei prodotti;
- `FinalData.s`: funzione per la stampa dei risultati della pianificazione;
- `InsertionSort.s`: funzione per il primo ordinamento dei prodotti;
- `Intestazione.s`: scrittura dell'intestazione condizionale dell'output del programma;
- `menu.s`: file contenente la funzione per l'invocazione del menù utente;
- `main.s`: funzione principale del programma;

Alcuni stralci di codice significativi:

- Allocazione della memoria `main.s`:

```
130     malloc                  # alloca spazio per l'array
131
132     movl len, %eax
133     sal $2, %eax            # moltiplico per 4 (4byte per int) (perchè ho 4byte
            ogni riga/prodotto)
134     movl %eax, len
135
136     movl $45, %eax          # syscall per la malloc
137     xorl %ebx, %ebx         # alloco inizialmente 0byte
138     int $0x80
139
140     movl %eax, %esi         # salvo l'indirizzo di memoria in esi (zona di
            memoria puntata dalla malloc)
141
```

```
    addl len, %eax          # aggiungo len a %eax (buffer allocazione)

    movl %eax, %ebx         # salvo il risultato in ebx (lunghezza totale da
        allocare)
    movl $45, %eax          # syscall per la malloc
    int $0x80               # alloco la memoria - interrupt

    cmpl %esi, %eax         # compara nuovo puntatore col vecchio (se sono
        uguali non ha allocato)
    jle Error               # esi è l'indirizzo di memoria puntato dalla malloc
        (inizio array) mentre in eax c'è l'indirizzo di fine array

    movl %esi, ArrayPointer # se tutto funziona salvo l'indirizzo di memoria in
        ArrayPointer cosi lo posso usare in BufferToArray
```

- Insertion sort `insertionSort.s`:

```
      .section .data

    key .int 0
    base .int 0

.section .text
    .global insertionSort
    .type insertionSort, @function

insertionSort

    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %edi # edi = Arr[fine]
    movl 12(%ebp), %esi # esi = Arr[inizio]
    movl 16(%ebp), %ecx # Metodo di sorting (+3 Priorità, +2 Scadenza)

    popl %ebp

    movl %ecx, base

    xorl %ecx, %ecx # i = 0
    xorl %edx, %edx # j = 0
    addl base, %ecx # %ecx punta alla priorità/scadenza
    addl $4, %ecx  # %punta a 2 oggetto (1)

for

    cmpl %ecx, %edi
    jle endFor

    movl (%esi, %ecx, 1), %eax
    movb %al, key
    subl base, %ecx
    movl (%esi, %ecx, 1), %eax
    push %eax
    addl base, %ecx

    movl %ecx, %edx # j = i
    addl $-4, %edx # j = i - 1

while

    cmpl $0, %edx
    jl PrepEndWhile

    movb (%esi,%edx,1), %al

    cmpb %al, key
    jg PrepEndWhile

    subl base, %edx
    movl %edx, %eax
```

```
    addl $4, %eax
    movl (%esi ,%edx,1), %ebx
    movl %ebx, (%esi ,%eax,1)
    decl %edx

    cmpl $3, base
    je while

    decl %edx

    jmp while

PrepEndWhile

    incl %edx

    cmpl $3, base
    je endWhile

    incl %edx

endWhile

    popl %eax
    movl %eax, (%esi ,%edx, 1)

    addl $4, %ecx

    jmp for

endFor

    ret
```

- Bubble sort `equalSort.s`:

```
      .section .data

base .byte 0                # EDF or HPF
flag .byte 0                # 1 -> swap 0 -> no swap

.section .text
    .global EqualSort
    .type EqualSort, @function

EqualSort

    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %edi      #len
    movl 12(%ebp), %esi     #pointer
    movl 16(%ebp), %ecx     #Base

    popl %ebp               #restore stack

    movb %cl, base          #salvo base (EDF or HPF)
    movl %ecx, %eax         # i
    movl %eax, %ebx         # j
    addl $4, %ebx           # j + 1

loop

    cmpl %eax, %edi         #len <= i (uscita)
    jle endLoop

    xorl %ecx, %ecx
    xorl %edx, %edx

    movb (%esi, %eax, 1), %cl
    movb (%esi, %ebx, 1), %dl
```

```
165
166     cmpb %cl, %dl               #confronto i e j
167     je Check                    #se sono uguali vado a check
168
169     jmp next
170
171 Check
172
173     cmpb $3, base               #se base = HPF (priorità) -> decrement
174     je decrement
175
176 increment                      # guardo priorità
177
178     incl %eax                   #incremento i
179     incl %ebx                   #incremento j
180
181     movb (%esi, %eax, 1), %cl       # preparo il confronto dei prossimi valori
182     movb (%esi, %ebx, 1), %dl
183
184
185     subl $3, %eax           # torno a puntare l'inizio dell'elemento
186     subl $3, %ebx           # =
187
188     cmpb %cl, %dl           # confronto i e j
189     jl reset
190     je SetDurata
191
192     movl (%esi, %eax, 1), %ecx
193     movl (%esi, %ebx, 1), %edx
194
195     movl %ecx, (%esi, %ebx, 1)
196     movl %edx, (%esi, %eax, 1)
197
198     movb $1, flag
199
200     jmp reset
201
202 decrement                  # guardo scadenza
203
204     decl %eax
205     decl %ebx
206
207     movb (%esi, %eax, 1), %cl
208     movb (%esi, %ebx, 1), %dl
209
210
211     subl $2, %eax
212     subl $2, %ebx
213
214     cmpb %dl, %cl
215     jg reset
216     je SetDurata
217
218     movl (%esi, %eax, 1), %ecx
219     movl (%esi, %ebx, 1), %edx
220
221     movl %ecx, (%esi, %ebx, 1)
222     movl %edx, (%esi, %eax, 1)
223
224     movb $1, flag
225
226     jmp reset
227
228 SetDurata        # se scadenza uguale priorità uguali, controllo la durata
229
230     addl $1, %eax    # incremento così vado da ID a durata
231     addl $1, %ebx
232
233 Durata
234
```

```
    movb (%esi, %eax, 1), %cl
    movb (%esi, %ebx, 1), %dl


    subl $1, %eax
    subl $1, %ebx

    cmpb %cl, %dl
    jge reset

    movl (%esi, %eax, 1), %ecx
    movl (%esi, %ebx, 1), %edx

    movl %ecx, (%esi, %ebx, 1)
    movl %edx, (%esi, %eax, 1)

    movb $1, flag

reset

    addb base, %al     # resetto base
    addb base, %bl     # resetto base

next

    cmpb $1, flag
    je Redo

    addl $4, %eax
    addl $4, %ebx
    jmp loop

Redo                   # se ho fatto uno swap, ripeto il confronto

    movl $0, flag
    movl base, %ecx
    movl %ecx, %eax
    movl %eax, %ebx
    addl $4, %ebx

    jmp loop

endLoop

    ret
```

- Loop scrittura `FinalData.s`:

```
      .section .data

PenaltyMsgLen .byte 9
TimeMsgLen .byte 13

PenaltyMsg .string "Penalty: "
TimeMsg .string "Conclusione: "

timeAscii .byte 4
penaltyAscii .byte 6


.section .text
    .globl finalData
    .type finalData, @function

finalData

    pushl %ebx #salvo la penalita'
    pushl %eax #salvo il tempo
    addl %edx, %edi # aggiungo la lunghezza del buffer all'inizio del buffer
        per avere il primo spazio libero
    xorl %ecx, %ecx #per sicurezza azzero ecx
```

```
152
153     movl $TimeMsg, %esi
154     movb TimeMsgLen, %cl
155     cld                         # flag reset direction
156     LoopTime                   # ctrl c e ctrl v
157         lodsb                   # carica byte da %esi ad %al
158         stosb                   # scrive byte da %al a %edi
159         loop LoopTime           # decrementa %cl e se non è zero salta a LoopTime
160
161
162 getTime        # Tempo in numeri in %ebx e dove scrivere in %edi
163
164     popl %eax   # Riprendo dalla stack il tempo
165     movl $10, %ebx
166     leal timeAscii+3, %esi    #un semplicissimo btoa che trasforma il tempo in
            ascii
167     movl $10, (%esi)
168
169 loopTime_B
170
171     xorl %edx, %edx          # pulisco edx
172     divl %ebx                # divido per prendere il resto
173
174     addb $48, %dl            #aggiungo 48 per trasformare il numero in ascii
175     decl %esi                #decremento il puntatore che fino a questo punto
            punta a 10 (vedi sopra)
176     movb %dl, (%esi)
177
178     test %eax, %eax
179     jz BufferTime
180
181     jmp loopTime_B
182
183 BufferTime # copio il buffer in %edi(puntatore al buffer)
184
185     movb (%esi), %al
186     cmpb $10, %al
187     je next
188     movb %al, (%edi)
189
190     incl %edi
191     incl %esi
192
193     jmp BufferTime
194
195 next  #finito il tempo uso il tappo \n
196
197     movb $10, (%edi)
198     incl %edi
199
200     xorl %ecx, %ecx
201
202     movl $PenaltyMsg, %esi
203     movb PenaltyMsgLen, %cl
204     cld                                 # ctrl c e ctrl v
205     LoopPenalty
206         lodsb
207         stosb
208         loop LoopPenalty
209
210 getPenalty
211
212     popl %eax # Riprendo dalla stack la penalita'
213     movl $10, %ebx
214     leal penaltyAscii+5, %esi
215     movl $10, (%esi)
216
217 loopPenalty
218
219     xorl %edx, %edx
```

```
220     divl %ebx
221
222     addb $48, %dl
223     decl %esi
224     movb %dl, (%esi)    # 'time loop' ma per la penalita'
225
226     test %eax, %eax     # se è 0 esco
227     jz BufferPenalty    # altrimenti continuo
228
229     jmp loopPenalty
230
231 BufferPenalty
232
233     movb (%esi), %al
234     cmpb $10, %al
235     je end
236     movb %al, (%edi)
237
238     incl %edi
239     incl %esi
240
241     jmp BufferPenalty
242
243 end
244
245     movb %al, (%edi)      #aggiungo \n
246
247     ret
```

In allegato si troveranno i seguenti dataset per testare il programma:

- `Makefile`: file per la compilazione del programma;
- `EDF.txt`: file di input con penalità uguale a zero con EDF e maggiore di zero con HPF;
- `BOTH.txt`: file di input con penalità uguale a zero con entrambi gli algoritmi;
- `NONE.txt`: file di input con penalità maggiore di zero con entrambi gli algoritmi.

1.5 Scelte progettuali

1.5.1 Ouput su file / funzioni extra-specifiche

Durante la programmazione del software sono state implementate alcune funzionalità aggiuntive:

- Salvataggio dei risultati della pianificazione su file;
- Visualizzazione del menù utente multi-scelta;
- Possibilità di lanciare il software senza specificare un file di output.
- Modifica a runtime di file di input e output.
- Visualizzazione di un messaggio di errore nel caso di file di output non non specificato.

1.5.2 Algoritmi di sorting

Per l'ordinamento dei prodotti, sono stati implementati due algoritmi di sorting differenti:

- `Sorting principale`: ordinamento basato su algoritmo 'insertion sort' per il confronto diretto tra il valore indicato dalla tipologia di pianificazione (EDF: deadline, HPF: priorità);
- `Sorting secondario`: ordinamento secondario basato su algoritmo di tipo 'bubble sort' per il confronto in caso di valore principale identico.

1.5.3 Limiti del software

Durante la fase di testing sono sorte alcune limitazioni del software:

- Il software non è in grado di gestire generalmete prodotti con valori maggiori di 256;
- Il software non è in grado di gestire file di input aventi dimensione maggiore di 1024 Byte.
- Il software non è in grado di gestire file di output aventi dimensione maggiore di 1024 Byte.