ARCHITETTURA DEGLI ELABORATORI - UNIVR

Elaborato ASSEMBLY, 2° Semestre-2024

Alessio Vivaldelli - VR502879
Ussama Badr Ezzamane - VR485687

Sommario

SCOPODELPROGETTO

Lo scopo del progetto è quello di realizzare un software che permette la pianificazione delle attività di un sistema produttivo. Ogni prodotto è caratterizzato da: un codice identificativo (1-127), da una durata(1-10), da una scadenza(rappresenta il tempo massimo in cui il prodotto dovrà essere completato) e da una priorità(1-5). Nello specifico, dato un file degli ordini (Ordini.txt) e data la possibilità all'utente la scelta dell'algoritmo di riordinamento (EDF oppure HPF), il software ./pianificatore dovrà generare l'ordine dei prodotti, specificando per ciascun prodotto l'unità di tempo in cui è pianificato l'inizio della produzione del prodotto. Inoltre, vengono stampati rispettivamente la conclusione la quale indica l'unità di tempo in cui è prevista la conclusione dell'ultimo prodotto pianificato, e la penalità. La penalità viene calcolata in base al ritardo ottenuto dalla differenza della scadenza del prodotto moltiplicato per la priorità del prodotto stesso

STRUTTURA DEL PROGETTO

Il programma 'pianificatore' è strutturato in diversi file che eseguono compiti specifici tramite funzioni. Nello specifico i files sono quelli riportati nell'immagine seguente.

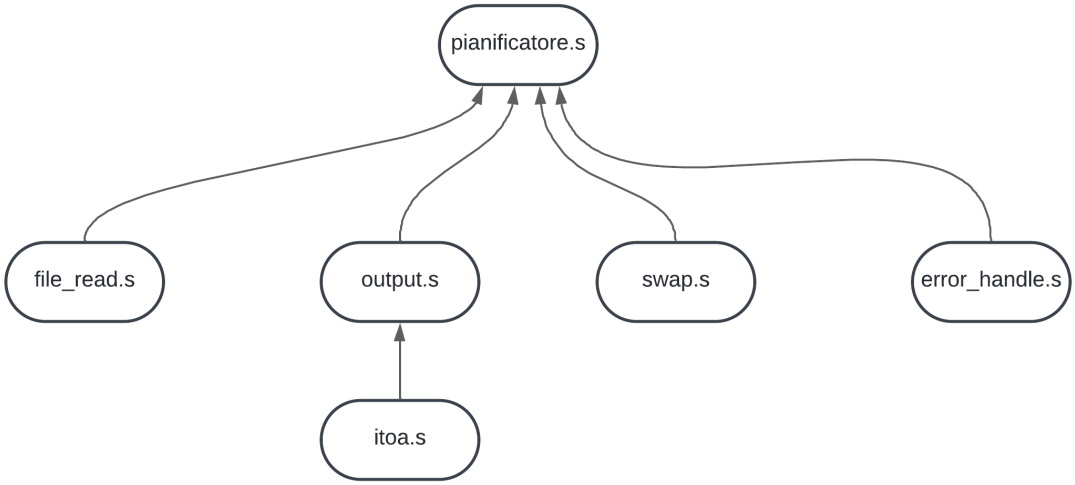

Figura 1 Struttura dipendenze del progetto

ERROR_HANDLE.S

Questo file, che è strutturato come una funzione, viene usato per gestire gli errori che si possono presentare lungo tutto il processo. Abbiamo deciso di raggruppare in un unico file i vari print a schermo poiché risulta più facile gestire gli errori semplicemente da un identificativo in un unico posto. Ad esempio, come si può

trovare nel file pianificatore a linea 173, quando il programma vede che non sono stati passati argomenti dalla riga di comando gestisce l'errore tramite 'error_handle':

```
movl $3, %eax
call error_handle
```

In questo caso è stato usato l'identificativo 3 che corrisponde a quell'errore.

La funzione è composta principalmente da una serie di 'if' che emulano un blocco 'switch case' per determinare quale ID ha chiamato la funzione. Dopodiché si fa un salto all'etichetta corrispondente che effettua la sys_write su 'stderr' e, dipende dal caso a caso, termina il programma con il codice di errore.

FILE_READ.S

All'interno di questo programma è presente una funzione che, data la stringa del filepath dell'ordine caricato in EBX, apre il file ed esegue il seguente processo per salvare i vari dati che ci sono dentro:

- **Salvataggio PC e base dei dati**
  - Quando la funzione viene chiamata come prima operazione viene salvato il 'Program Counter', puntato dal ESP, sul registro EBP. Questo è necessario poiché nella funzione viene aggiunto materiale alla stack quindi nel momento del 'ret' andrebbe in Segmentation Fault poiché interpreterebbe l'ultimo valore della stack come PC.
  - Il valore che abbiamo tolto dallo stack, lo usiamo come base dei nostri dati assegnandoli il valore –1. Facendo così possiamo capire quando siamo arrivati alla fine dei dati scorrendoli.
- **Conversione da caratteri ad intero:**
  - Per convertire i numeri scritti in ascii ad intero vengono iterati i vari caratteri fino al raggiungimento di una virgola/terminatore di stringa o nuova linea. Queste cifre vengono convertite singolarmente tramite una 'atoi' poi "unite" moltiplicando la cifra passata per dieci e sommata quella nuova.
- **Salvataggio dati**
  - Quando viene eseguita la conversione da stringa ad intero, questo dato viene poi aggiunto alla stack
- **Ripristino PC**
  - Essendo che all'inizio abbiamo salvato il PC in EBP, prima di ritornare aggiungiamo alla stack tale valore per far sì che la return funzioni correttamente

Di seguito è riportata il flowchart della sezione della funzione dove vengono convertiti e salvarti i dati.

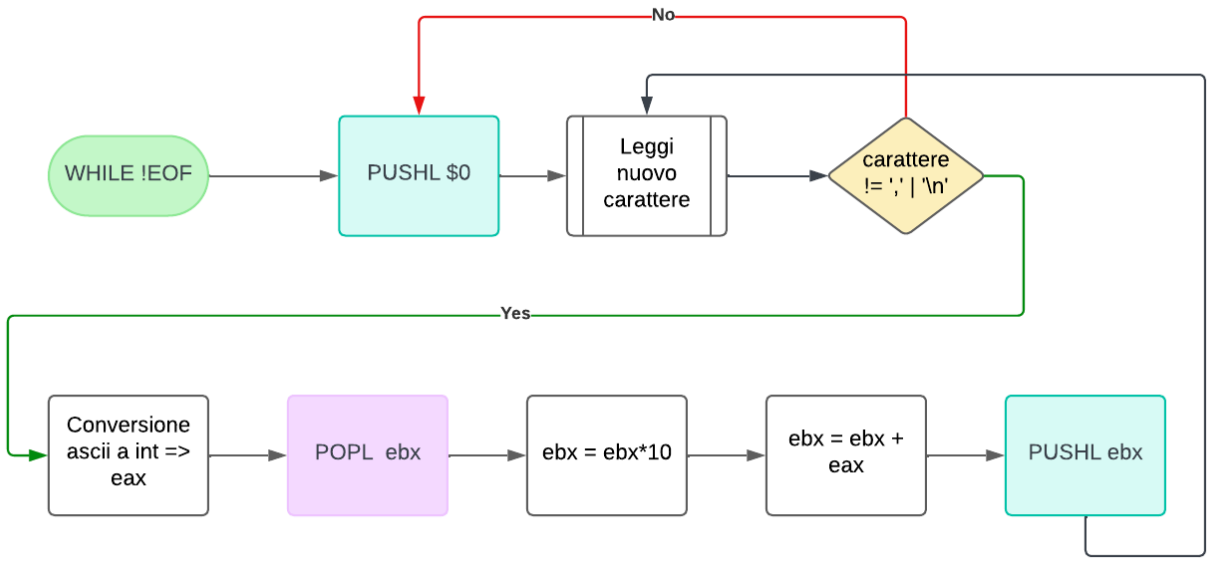

Figura 2 Flochart conversione e salvataggio dati

OUTPUT.S

La funzione output computa il risultato dell'algoritmo andando a leggere i dati dalla stack, che a questo punto sono già ordinati in base all'algoritmo selezionato in precedenza, ed esegue la stampa sullo stream giusto.

La voce conclusione viene calcolata inizializzando una variabile a 0 e incrementandola del valore di Durata di ogni ordine. Lo stesso valore viene utilizzato per la sezione di output che stampa "ID:Inizio", dove inizio è la variabile in questione prima dell'incremento.
La voce penalità invece viene calcolata tramite l'operazione:

$$\begin{cases} Penalità \mathrel{+}= (time - scadenza) * priorità \ \ se > 0 \\ Penalità \mathrel{+}= 0 \ altrimenti \end{cases}$$

Questa operazione viene effettuata solo se l'equazione risulta in un valore maggiore di zero. La variabile 'time' esprime il tempo in cui finisce l'elaborazione dell'ordine corrente.

Lo stream dove stampare il risultato dell'algoritmo è deciso dalla variabile fd, che è determinata come segue:
Inizialmente il valore di default di fd è 1 (stdout). La funzione è chiamata mettendo in EBX l'indirizzo di una stringa che fa riferimento al 3° parametro del programma principale. Come spiegato nella sezione del file principale, se tale parametro non esista il valore, che è un long, viene lasciato a 0.
All'inizio della procedura viene fatto un confronto con tale valore e se è diverso da 0 viene eseguito un codice che effettua la sys_open con i parametri:

- **1089 in ecx**: questo parametro definisce la modalità di apertura del file che equivale al 'a+'. Questa modalità, che di base definisce l'append ad un file, se

è seguita dal carattere ‘+’ definisce che se il file non esiste deve venire creato e di seguito aperto.

- **0644 in edx**: questo valore rappresenta i permessi del file qualora venga creato. Nello specifico questi indicano che il proprietario ha premessi di lettura e scrittura, mentre gli altri solo lettura.

Questi valori sono stati trovati andando nella libreria C ‘fcntl.h’ dove sono definiti i valori per le varie modalità di apertura dei file, in notazione ottale. Facendo la somma delle modalità che preferiamo otteniamo il risultato da mettere in ECX.
Di seguito è ripotato un esempio di come sono definiti i vari valori nella libreria:

```
#define O_ACCMODE     0003
#define O_RDONLY       00
#define O_WRONLY       01
#define O_RDWR         02
….
```

Quindi la chiamata di sistema risultante sarà:
“*open("file_path", O_WRONLY|O_CREAT|O_APPEND, 0644)*”

Se la sys_open viene eseguita con successo in eax troveremmo il file descriptor che verrà salvato nella variabile fd.

Quando vengono effettuate le varie sys_write nel registro ebx, che indica lo stram, mettiamo il valore di fd. Quindi se esiste il 3° parametro e la sys_open è avvenuta con successo scriverà su file, altrimenti il risultato sarà scritto su stdout.

Infine, per stampare i valori che sono salvati come interi, vengono convertiti in stringhe tramite la funzione itoa.

SWAP.S

Questa funzione permette di invertire due ordini successivi basandosi sulla posizione attuale del puntatore EBP. Che sarà usato per scorrere i vari ordini.

PIANIFICATORE.S

Questo è il file principale da dove vengono chiamate tutte le altre funzioni descritte precedentemente e dove avvengono altri processi.

Questo codice si occupa dei seguenti processi:

1. **Estrazione argomenti**
2. **Acquisizione da tastiera nel menu**
3. **Ordinamento degli ordini**

1. Estrazione argomenti

In questa sezione si estraggono dalla stack gli argomenti che si passano da terminale al programma. Nella seguente fase si controlla inizialmente che sia presente almeno un parametro, quello degli ordini, e in caso contrario gestisce l'errore tramite la funzione output precedentemente descritta.

Quando un valore viene preso dalla stack viene poi spostato in una variabile di tipo long, così facendo non dobbiamo allocare memoria per una stringa che contenga il percorso del file, ma solo l'indirizzo in memoria di dove è salvata.
Inizialmente questa variabile è inizializzata a 0 e, se esiste l'argomento, sovrascritta con il valore.

Sapendo che gli argomenti, quando un programma viene lanciato, sono posizionati nello stack in un ordine preciso, le prime due azioni che vengono fatte sono due 'popl' che permettono di rimuovere il numero di argomenti e il percorso del programma in questione. Così facendo i successivi 'popl' permetteranno di estrarre i parametri.

2. Acquisizione da tastiera nel menu

Quando il programma viene avviato, all'utente verrà mostrato un menù dove potrà scegliere tra i due algoritmi di pianificazione (EDF e HPF) oppure uscire dal programma. Il codice effettua un controllo sull'input inserito e in caso mostra l'errore opportuno.

Con la selezione dell'algoritmo il codice andrà a salvare tale scelta in una variabile dedicata poiché servirà successivamente nella sezione di sorting.

3. Ordinamento degli ordini

In questa parte del codice avviene il processo per cui gli ordini, precedentemente posizionati sulla stack dalla funzione 'read_file', vengono ordinati in funzione dell'algoritmo selezionato.

Essendo che i dati sono stati messi sulla stack effettuando delle 'pushl' e nell'ordine inverso rispetto a quello da consegna, per accedere ai valori di interesse dovremmo effettuare i seguenti offset rispetto all'esp:

- ID: 12
- Durata: 8
- Scadenza: 4
- Priorità: 0

Per ordinare i valori abbiamo optato per un Bubble Sort principalmente per due motivi:

- Da vincolo progettuale gli ordini sono sempre minori di 10, per questo motivo l'algoritmo che si usa non influenza in maniera impattante sulle performance del programma.
- Questo algoritmo lavora sempre con valori consecutivi e non crea partizioni dell'array; quindi, fa sì che sia meno insidioso nella realizzazione.

Gli ordini vengono comparati tramite una funzione, tale funzione è decisa in base all'algoritmo selezionato precedentemente. Quindi sono state scritte due funzioni che comparano i campi in questione e, se necessario, effettuano lo scambio tramite la funzione swap.

Di seguito è riportato il codice C analogo a quello assembly:

```
/*
* EDF
*/
int EDF_compare(Order order_1, Order order_2)
{
    if(order_1.scadenza > order_2.scadenza){return 1;}
    else if((order_1.scadenza == order_2.scadenza) && order_1.priority < order_2.priority){return
2;}
    else{return 0;}
}

/*
* HPF
*/
int HPF_compare(Order order_1, Order order_2)
{
    if(order_1.priority < order_2.priority){return 1;}
    else if((order_1.priority == order_2.priority) && (order_1.scadenza
>     order_2.scadenza)){return 2;}
    else{return 0;}
}

printf("Start sorting\n");
int flag = 1;
while(flag != 0)
{
    flag = 0;
    for (int i = 0; i < num-1; i++)
    {
        int comp = compare(orders[i], orders[i+1]);
        if(comp == 1)
        {
            Order tmp = orders[i+1];
```

```
            orders[i+1] = orders[i];
            orders[i] = tmp;
            flag = 1;
        }
        if(comp == 2)
        {
            Order tmp = orders[i+1];
            orders[i+1] = orders[i];
            orders[i] = tmp;
            flag = 1;
        }
    }
}printf("finished\n");
```

Quando l’algoritmo termina si passa alla sezione successiva gestita della funzione ‘output’.

RIEPILOGO

Di seguito è possibile vedere un flowchart che fa una sintesi del funzionamento totale del progetto.

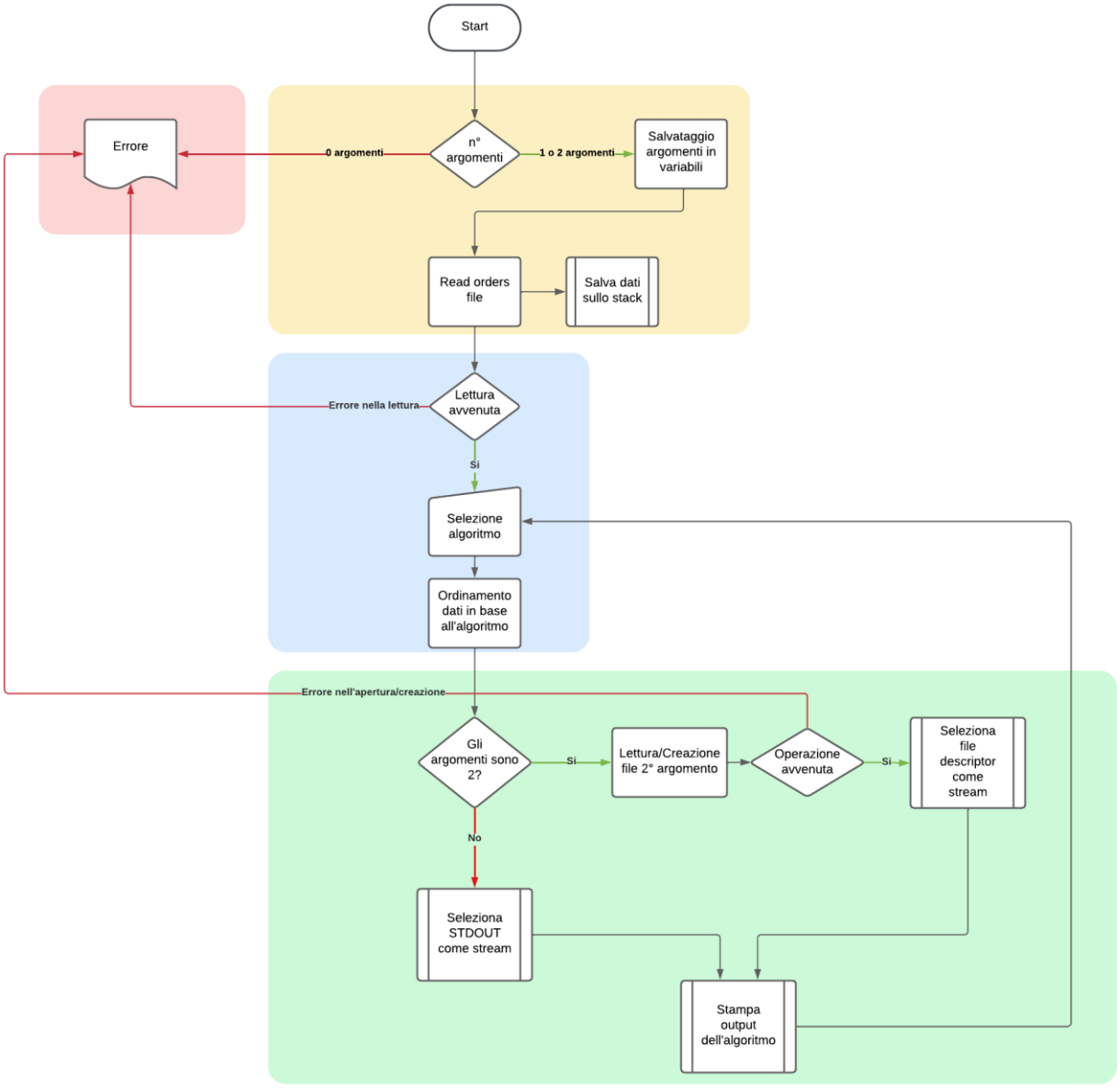

Figura 3 Flowchart generale del progetto

SCELTE PROGETTUALI

La scelta dell’uso dello stack è stata vincolata dal fatto che, avendo un massimo di 10 valori per file, non abbiamo ritenuto necessario l’utilizzo dell’heap per il salvataggio dei valori, poiché non presenta un rischio di uno stack overflow.

Il progetto non è vincolato con il massimo di 10 ordini, ma può gestirne un numero non definito. Comunque, può succedere che, se sono inseriti un numero alto di ordini, la stack può causare problemi dato dal fatto che ha una dimensione massima.

Abbiamo inoltre ritenuto opportuno di separare le varie funzioni su più file per rendere il codice più leggibile, pulito e facilitare le sessioni di debug.