Architettura degli Elaboratori: ASM scheduler - elaborato

Simone Di Maria (VRXXXXXXXX), Pietro Secchi (VRXXXXXXXX)

23 luglio 2024

Indice

1 Specifiche del Progetto

Viene richiesto di sviluppare un software per la pianificazione delle attività di un sistema produttivo in codice **Assembly x86 con sintassi AT&T** .
Il sistema produttivo è in grado di produrre prodotti diversi, ma può produrre un solo prodotto alla volta in modo sequenziale. La produzione è suddivisa in slot temporali uniformi e solo un prodotto può essere in produzione.
Ogni prodotto o "*task*" è caratterizzato da quattro valori interi (proprietà della task):

- **Identificativo**: Il codice identificativo del prodotto da produrre. Il codice può andare da 1 a 127.
- **Durata**: Il numero di slot temporali necessari per completare il prodotto. La produzione di ogni prodotto può richiedere da 1 a 10 slot temporali.
- **Scadenza**: Il tempo massimo, espresso come numero di unità di tempo, entro cui il prodotto dovrà essere completato. La scadenza di ciascun prodotto può avere un valore che va da 1 a 100.
- **Priorità**: Un valore da 1 a 5, dove *1 indica la priorità minima e 5 la priorità massima.* Il valore di priorità indica anche la penalità che l'azienda dovrà pagare per ogni unità di tempo necessaria a completare il prodotto oltre la scadenza.

Per ogni prodotto completato in ritardo rispetto alla scadenza indicata, l'azienda dovrà pagare una penale in Euro pari al valore della priorità del prodotto completato in ritardo, moltiplicato per il numero di unità di tempo di ritardo rispetto alla sua scadenza.
In altre parole, se una task T con Durata T_D, priorità T_P e scadenza all'unità di tempo T_{Exp}, viene messa in produzione all'unità di tempo x, la penalità P è calcolata come segue:

$$P = max(0,\ (x + T_D - T_{Exp})) * T_P$$

Le task verranno caricate allo scheduler tramite lettura da file. Le specifiche di I/O sono delineate nel paragrafo seguente. Una volta letto il file, il programma mostrerà il menu principale che chiede all'utente quale algoritmo di pianificazione dovrà usare. L'utente potrà scegliere tra i seguenti due algoritmi di pianificazione:

1. **Earliest Deadline First (EDF)**: si pianificano per primi i prodotti la cui scadenza è più vicina; in caso di parità nella scadenza, si pianifica il prodotto con la priorità più alta.
2. **Highest Priority First (HPF)**: si pianificano per primi i prodotti con priorità più alta; in caso di parità di priorità, si pianifica il prodotto con la scadenza più vicina.

L'utente dovrà inserire il valore 1 per chiedere al software di utilizzare l'algoritmo EDF, ed il valore 2 per chiedere al software di utilizzare l'algoritmo HPF.
Una volta elaborate le task, lo scheduling delle task a seconda dell'algoritmo scelto verrà stampato a video.

2 Struttura del progetto

2.1 Struttura generale

Il codice Assembly è compilato in binario eseguibile formato ELF (Executable and Linkable Format) con il comando `make` e il file `Makefile`. Quest'ultimo si occuperà dell'Assemblaggio e del Linkaggio del codice, utilizzando rispettivamente i tool `as` ed `ld` della collezione GNU Binutils.
Il binario compilato dovrà essere eseguito mediante la seguente linea di comando:

./pianificatore <percorso del file delle tasks>

Ad esempio, se il comando dato fosse:

./pianificatore Ordini.txt

il software caricherà gli ordini dal file *Ordini.txt*.
Il file delle tasks dovrà avere un prodotto per riga, con tutti i parametri separati da virgola. Ad esempio, se i prodotti fossero:

- **Identificativo**: 1; **Durata**: 10; **Scadenza**: 12; **Priorità**: 4;
- **Identificativo**: 2; **Durata**: 17; **Scadenza**: 32; **Priorità**: 5;

il file dovrebbe contenere le seguenti righe:

1,10,12,4
2,7,32,1

Il software leggerà duque riga per riga il file fornito come argomento, allocando ogni task in memoria. Per favorire la lettura da file si è deciso di leggere la grandezza di una pagina virtuale di memoria alla volta, che in sistemi Linux con processore x86-64 sarà di 4096 byte.
Nel caso in cui dopo aver letto una n-esima pagina di memoria, venga troncata la lettura di una riga (i.e. di una task), viene utilizzata la syscall `lseek` per riposizionare il puntatore del buffer del file all'inizio della riga.

Scelta Progettuale n. 1

Per favorire la lettura da file, viene letta una pagina di memoria virtuale alla volta, riposizionando il puntatore del buffer del file alla posizione corretta, utilizzando la syscall `lseek`, quando necessario.

Per una complessità computazionale ottimale nella creazione di una lista ordinata secondo l'algoritmo di ordinamento dato, le tasks vengono allocate all'interno di nodi di una Linked List. Nello specifico, è stata implementata nel codice la struttura dati di una ***Linked List Circolare Doppiamente Linkata*** (Circular Doubly Linked List) per rendere possibile allocare ed ordinare le tasks nello stesso momento.
Tale approccio è risultato essere un ottimo compromesso tra efficienza computazionale ed efficienza in termini di memoria. Questa scelta progettuale è approfondita nel paragrafo dedicato.

Scelta Progettuale n. 2

Per un'alta scalabilità, le tasks vengono processate ed ordinate con la struttura dati di una Linked List Circolare Doppiamente Linkata.

Infine, i nodi ordinati, allocati in memoria, vengono decodificati per poter essere stampati a video.

Di seguito è riportata lo schema generale della struttura appena descritta.

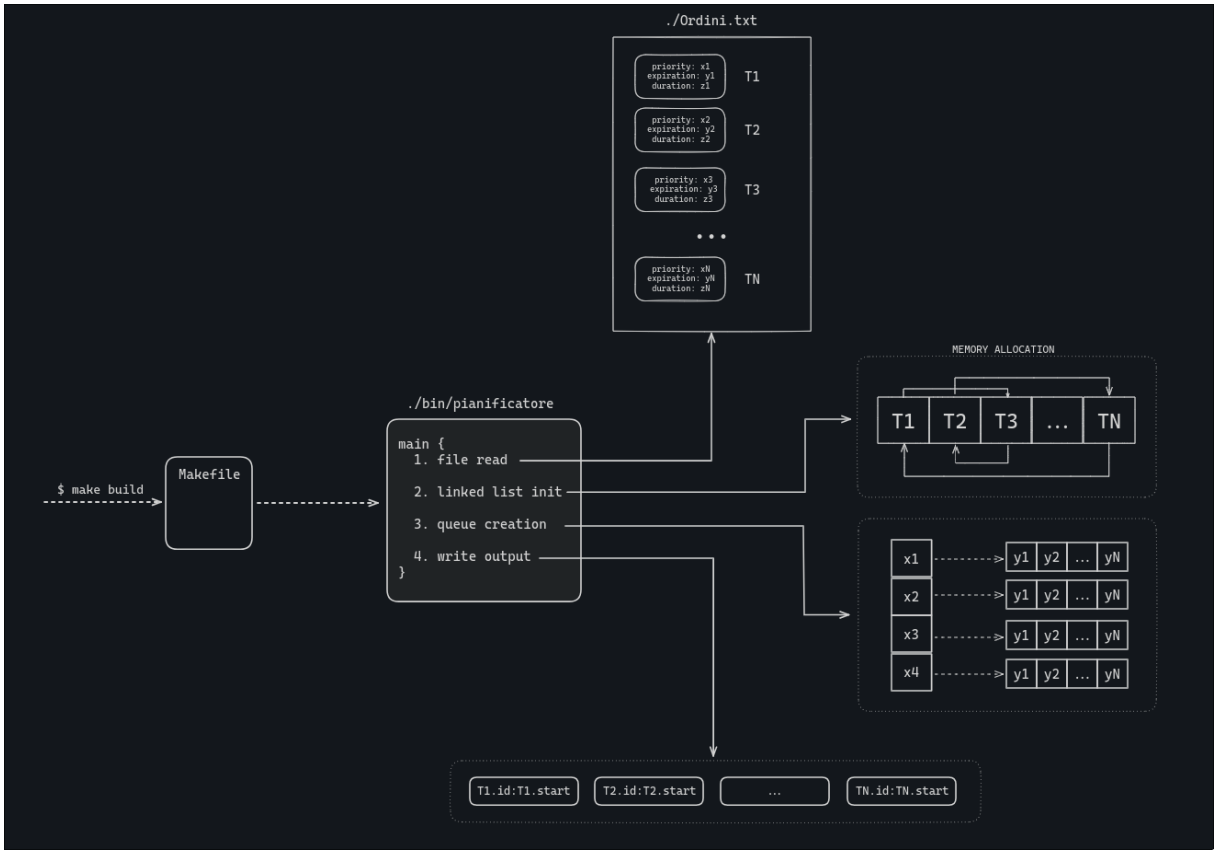

Figura 1: Scheduler structure

2.2 Queue e Linked List

Come precedentemente scritto, ogni task letta da file viene trattata come nodo di una Linked List. In particolare, si è deciso di implementare una Linked List Doppiamente Linkata, che permette la facile iterazione della lista in entrambe le direzioni.
Quest'ultima è stata anche definita come Circolare: la "***head***" della Linked List viene linkata sia con il primo nodo, che con l'ultimo nodo, per renderli facilmente accessibili senza dover iterare tutta la lista.
Queste caratteristiche della Linked List risultano efficaci in diverse situazioni, ad esempio leggere la lista in ordine contrario, accedere al primo ed ultimo nodo senza iterare la lista completa, etc. Tuttavia, la creazione di una singola Linked List non è efficente in determinati casi limite: Nel caso in cui due task abbiano pari priorità (o pari scadenze), bisogna dare la precedenza alla task con scadenza più vicina (o priorità più alta). Questo risulta un problema nel caso in cui, ad esempio, si presentassero un numero grande N di task con priorità (o scadenza) uguali, poichè richiederebbe di dover iterare tutte le N task.
La soluzione implementata per ovviare a questo problema, è stata l'introduzione di una seconda struttura dati complementare alla Linked List: la ***Queue***.
La Queue è a sua volta una Linked List, in cui però non vengono salvate tutte le informazioni delle tasks. I nodi della Queue sono composti da:

1. **Il valore dalle priorità** (o scadenze) **univoche** (nel caso in cui due task abbiano priorità (o scadenza) P uguale, nella queue non ci saranno due nodi con priorità uguale P, bensì uno solo).
2. **Un puntatore ad un'altra Linked List** contenente tutti i nodi che hanno tale priorità P (o scadenza) uguale, ordinati però in base alla scadenza (o priorità).

Sia la Queue, che le rispettive Linked List, sono ordinate in ordine crescente.
Di seguito è riportata la rappresentazione grafica di tale struttura dati combinata (Queue e rispettive Linked List).

Scelta Progettuale n. 3

Per ottimizzare la gestione delle task e migliorare l'efficienza nella ricerca e ordinamento, è stata implementata una **Linked List Circolare Doppiamente Linkata**. Questa struttura permette una navigazione bidirezionale e facilita l'accesso immediato ai nodi iniziale e finale.

Scelta Progettuale n. 4

Per affrontare i casi di task con priorità o scadenze identiche, è stata introdotta una **Queue** complementare. La Queue contiene nodi con priorità o scadenze uniche e puntatori a Linked List secondarie, che memorizzano task con valori uguali, ordinati secondo l'altro criterio. Questo approccio riduce il tempo di ricerca e migliora l'efficienza dell'algoritmo di scheduling.

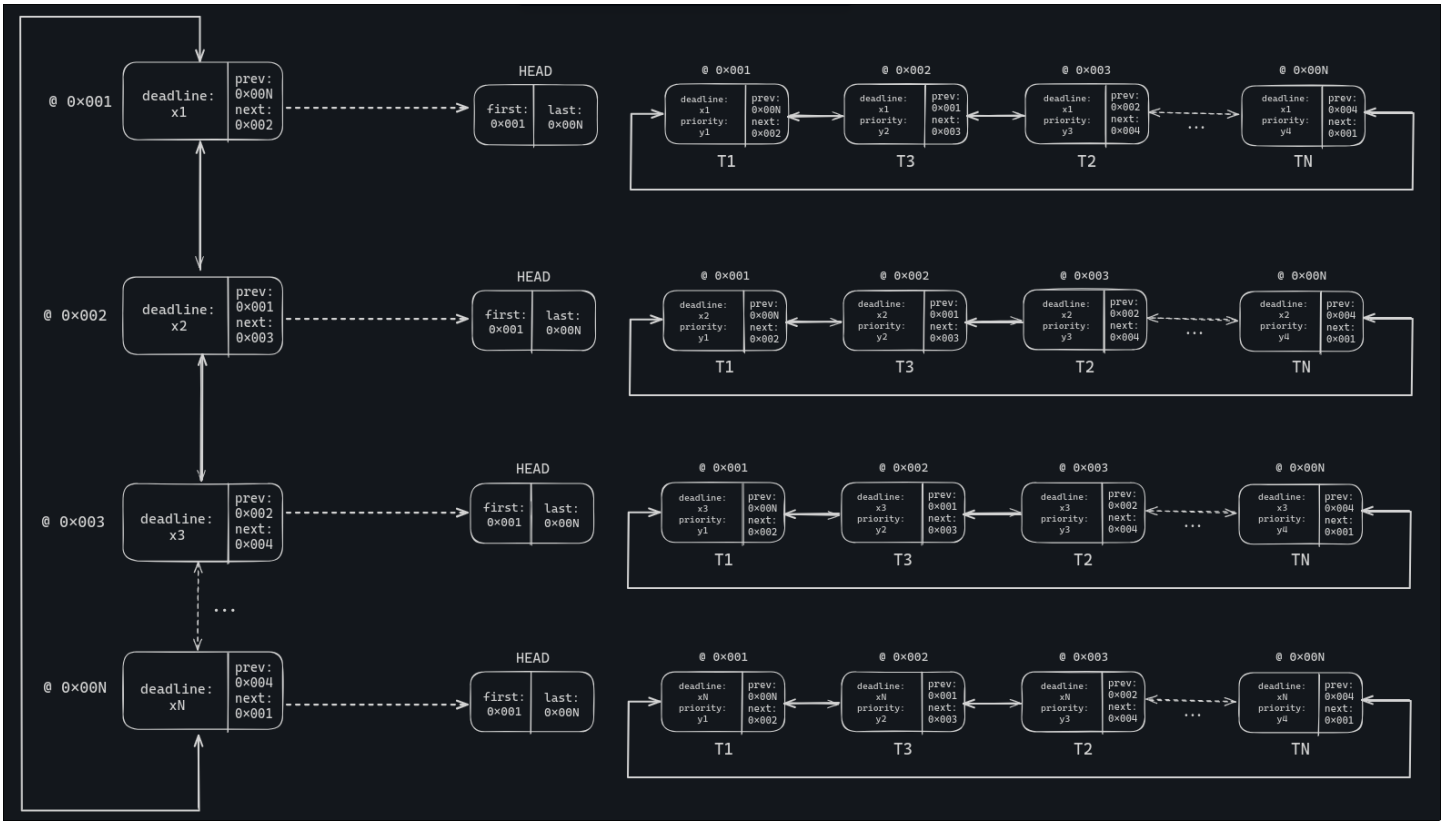

Figura 2: Scheduler structure

2.3 Struttura del codice sorgente

La struttura del progetto a livello di codice sorgente è organizzata come segue:

Struttura del codice sorgente

```
$ tree .
.
|-- src
|   |-- common_utils.s
|   |-- constants.s
|   |-- main.s
|   |-- queue.s
|   |-- readfile.s
|   |-- sll.s
|   |-- sll_utils.s
|   |-- task_utils.s
|   |-- ui.s
|-- Makefile
|-- obj
|   |-- common_utils.o
|   |-- constants.o
|   |-- main.o
|   |-- queue.o
|   |-- readfile.o
|   |-- sll.o
|   |-- sll_utils.o
|   |-- task_utils.o
|   |-- ui.o
|-- bin
|   |-- scheduler
|-- scheduler.py
|-- TestCaseGenerator.py
|-- test_cases.txt
|-- release
    |-- VRXXXXXXX_VRXXXXXXX.tar.gz
|-- report
|   |-- assets
|       |-- linkedlist.md
|       |-- struttura_progetto.md
|   |-- REPORT.pdf
|-- resources
|   |-- morton2007.pdf
|   |-- OUTLINE.pdf
```

Di seguito sono invece riportate le documentazioni dei rispettivi moduli.

2.3.1 `constants.s` module

Contiene costanti globali utilizzate come simboli nel codice assembly.

Syscalls

- `SYS_EXIT`: 1
- `SYS_READ`: 3
- `SYS_WRITE`: 4
- `SYS_OPEN`: 5
- `SYS_CLOSE`: 6
- `SYS_LSEEK`: 19
- `SYS_BRK`: 45

File Descriptors

- `STDIN`: 0
- `STDOUT`: 1
- `STDERR`: 2

Miscellaneous

- `PAGE_SIZE`: 4096
- `SEEK_CUR`: 1
- `O_RDONLY`: 0
- `LINE_FEED_ASCII`: 10
- `COMMA_ASCII`: 44
- `COLON_ASCII`: 58

2.3.2 `main.s` module

Entrypoint del progetto.
Legge file da linea di comando e chiama funzioni dei vari moduli. Una volta completata l'esecuzione, stampa a video e/o in file, chiude i file ed esce.

2.3.3 `common_utils.s` module

Function Name	Arguments	Returns	Description
`atoi`	`ebx: ascii_str`	`eax: integer`	Converts an ASCII string to its decimal representation.
`itoa`	`eax: integer`, `ecx: fd`	`eax: ascii_str`	Converts an integer to its ASCII representation and writes it to the file descriptor.
`itoa_to_buffer`	`ebx: integer`	`ebx: buffer`	Converts an integer to its null-terminated ASCII representation in the buffer.
`open_file`	`ebx: filename`	`eax: fd`	Opens a file and returns the file descriptor.
`print_buffer`	`eax: buffer`, `ebx: length`, `ecx: fd`	-	Prints the buffer to the given file descriptor.
`print_buffer_no_length`	`eax: buffer`, `ebx: length`, `ecx: fd`	-	Prints the buffer to the file descriptor without the length.
`find_nullbyte`	`eax: buffer`	`ecx: buffer_length`	Finds the null byte in the buffer and returns its length.
`copy_buffer_to_buffer`	`ebx: source_buffer`, `ecx: dest_buffer`, `esi: buffer_length`	-	Copies the source buffer to the destination buffer.

2.3.4 `ui.s` module

Function Name	Arguments	Returns	Description
`start_ui`	-	-	Starts the user interface.
`print_menu`	-	-	Prints the user interface menu.
`handle_user_input`	-	`ebx: user_choice`	Reads the user input and sets the `user_choice` variable.
`ask_for_input`	-	-	Prints the prompt for user input.

2.3.5 `readfile.s` module

Function Name	Arguments	Returns	Description
`init_file`	`ebx: filename`	`eax: fd`	Opens the file with the given filename, returns the file descriptor in `eax`.
`lseek`	`eax: file_buffer_position, ebx: fd`	`eax: lseek`	Moves the file pointer of the given file descriptor to the given file buffer position.
`read_tasks`	-	`eax: buffer`	Reads the file line by line, returns the buffer with the file content.
`decode_nodes`	`ebx: buffer, ecx: bytes_read`	-	Decodes the nodes from the file buffer, returns the decoded nodes in `buffer_nodes`.
`decode_node`	`ebx: buffer`	-	Decodes the nodes from the file buffer, returns the decoded nodes in `buffer_nodes`.
`get_broken_node`	`ebx: nodes_buffer, ecx: bytes_read`	`eax: bytes_to_lseek`	Returns the number of bytes to go back to resume decoding nodes from the file.

2.3.6 `sll.s` module

Function Name	Arguments	Returns	Description

`add_to_list`	`eax: list_head,`
	`ebx: value,`
	`ecx: priority`
		`eax: list_head`	Adds a new node to the linked list, sorted by the given type (0: ascending, 1: descending).

`merge_lists`	`eax: list1,` `ebx: list2`	`eax: merged_list`	Merges two linked lists into one, sorted by the given type (0: ascending, 1: descending).

`print_list`	`eax: list,`
	`ebx: fd,`
	`esi: algorithm`
		`eax: list`	Prints the list in id:time format in the specified order (0: descending, 1: ascending) to the file descriptor.

`check_if_first`	-	-	Checks if the new node is the first in the list and updates the first node if necessary.
`check_if_last`	-	-	Checks if the new node is the last in the list and updates the last node if necessary.

`list_to_buffer`	`eax: list_head,`
	`ebx:`
	`start_node_index,`
	`ecx: order`
		`edx: buffer,`
		`ebx: last_node_index`
			Converts the linked list to a buffer in the specified order (0: descending, 1: ascending).

2.3.7 `sll_utils.s` module

Function Name	Arguments	Returns	Description
`init_list`	`ecx: task_priority,` `edx: task_value`	`eax: head_addr`	Initializes a linked list, creating a head and first node, setting it as the last node, and setting its priority and value.
`allocate_node`	-	`eax: node_addr`	Allocates memory for a node in the linked list.
`allocate_head`	-	`eax: head_addr`	Allocates a head node for the linked list.

Function Name	Arguments	Returns	Description
insert_node	eax: prev_node_addr, ebx: new_node_addr, ecx: next_node_addr	eax: head_addr	Inserts an allocated node between two nodes.
compare_nodes	ecx: node1_addr, edx: node2_addr	eax: comparison	Compares two nodes by their priority.
set_first_node	eax: list_head, ebx: first_node_addr	eax: head_addr	Sets the first node address in the head.
set_last_node	eax: list_head, ebx: last_node_addr	eax: head_addr	Sets the last node address in the head.
set_next_node	eax: node_addr, ebx: next_node_addr	eax: node_addr	Sets the next node pointer in the given node.
set_prev_node	eax: node_addr, ebx: prev_node_addr	eax: node_addr	Sets the previous node pointer in the given node.
set_next_and_prev_node	eax: node_addr, ebx: next_node_addr	eax: node_addr	Sets the next node pointer in the given node and the previous node pointer in the next node, linking them bidirectionally.
set_node_value	eax: node_addr, ebx: value	eax: node_addr	Sets the value at the given node address.
set_node_priority	eax: node_addr, ebx: priority	eax: node_addr	Sets the priority value at the given node address.
get_first_node	eax: list_head	ebx: first_node_ptr	Retrieves the first node address from the list head.
get_first_node_ptr	eax: list_head	eax: first_node_addr	Gets the pointer to the first node from the list head.
get_last_node	eax: list_head	ebx: last_node_ptr	Retrieves the last node address from the list head.
get_last_node_ptr	eax: list_head	eax: last_node_addr	Gets the pointer to the last node from the list head.
get_next_node_ptr	eax: node_addr	eax: next_node_addr	Gets the pointer to the next node in the linked list.
get_next_node_address	eax: node_addr	ebx: next_node_addr	Gets the next node address from the given node address.
get_prev_node_ptr	eax: node_addr	eax: prev_node_addr	Gets the pointer to the previous node in the linked list.
get_prev_node_address	eax: node_addr	ebx: prev_node_addr	Gets the previous node address from the given node address.

Function Name	**Arguments**	**Returns**	**Description**
`get_node_priority`	`eax: node_addr`	`ebx: priority`	Retrieves the priority value from the node.
`get_node_priority_ptr`	`eax: node_addr`	`eax: priority_addr`	Retrieves the pointer to the node's priority value.
`get_node_with_priority`	`eax: list_head,` `ebx: target_priority`	`eax: node_addr`	Searches for a node with the given priority in the list, returning its address or -1 if not found.
`get_node_data_ptr`	`eax: node_addr`	`eax: value_addr`	Retrieves the pointer to the node's data value.
`get_value_value`	`eax: node_addr`	`ebx: value`	Retrieves the value from the node.

2.3.8 `queue.s` module

Function Name	Arguments	Returns	Description

`init_queue`	`eax: task_id,`
	`ebx: task_duration,`
	`ecx: task_expiration,`
	`edx: task_priority,`
	`esi: algorithm`
		`eax: queue`	Initializes a new queue with given tasks and algorithm (0: LDF, 1: HPF).

`init_queue_from_buffer`	`ebx: buffer,` `esi: algorithm`	`eax: queue`	Initializes a queue from a buffer.

`create_task_to_queue`	`eax: task_id,`
	`ebx: task_duration,`
	`ecx: task_expiration,`
	`edx: task_priority,`
	`esi: queue`
		`eax: new_task_address`	Create a task to the queue using the provided task parameters.

`add_task_to_queue`	`eax: queue,` `ecx: task`	`eax: new_task_address`	Adds an already allocated task to the queue instead of constructing it by passing task parameters.
`add_tasks_to_queue_from_buf`	`eax: queue,` `ebx: buffer`	`eax: queue`	Adds multiple tasks to the queue from a buffer.
`get_queue_algo`	`eax: queue`	`ebx: algorithm`	Retrieves the pointer to the selected algorithm for the queue.
`get_queue_algo_value`	`eax: queue`	`ebx: algorithm`	Retrieves the value of the selected algorithm for the queue.
`get_first_ll_ptr_from_queue`	`eax: queue`	`ebx: list_address`	Retrieves the address of the list in the queue.
`get_first_ll_addr_from_queue`	`eax: queue`	`ebx: *list_address`	Retrieves the value of the address of the list in the queue.
`set_queue_list_address`	`eax: queue`	`ebx: list_address`	Sets the address of the list in the queue.
`set_queue_algo`	`eax: queue`	`ebx: algorithm`	Sets the algorithm for sorting the queue.
`allocate_queue`	-	`eax: queue`	Allocates a new queue.
`queue_to_list`	`eax: queue,` `esi: algorithm`	`eax: list`	Transforms a queue in memory into a linked list.

2.3.9 `task_utils.s` module

Function Name	Arguments	Returns	Description
`allocate_task`	-	`eax: task_addr`	Allocates memory for a new task and returns its address.

`create_task`	`eax: task_id,`
	`ebx: task_duration,`
	`ecx: task_expiration,`
	`edx: task_priority`
		`eax: task`	Creates a new task with the specified parameters.

`print_task_details`	`eax: task`	-	Prints the details of the specified task to STDOUT.
`create_task_from_buffer`	`ebx: buffer`	`eax: task`	Creates a new task using the data from the specified buffer.
`get_task_id_ptr`	`eax: task`	`eax: task_id`	Returns the pointer to the task ID.
`get_task_priority_ptr`	`eax: task`	`eax: task_priority`	Returns the pointer to the task priority.
`get_task_expiration_ptr`	`eax: task`	`eax: task_expiration`	Returns the pointer to the task expiration.
`get_task_duration_ptr`	`eax: task`	`eax: task_duration`	Returns the pointer to the task duration.
`get_task_id_value`	`eax: task`	`ebx: *task_id`	Returns the task ID.
`get_task_priority_value`	`eax: task`	`ebx: *task_priority`	Returns the task priority.
`get_task_expiration_value`	`eax: task`	`ebx: *task_expiration`	Returns the task expiration.
`get_task_duration_value`	`eax: task`	`ebx: *task_duration`	Returns the task duration.
`set_task_id`	`eax: task,` `ebx: task_id`	`eax: task`	Sets the task ID.
`set_task_priority`	`eax: task,` `ebx: task_priority`	`eax: task`	Sets the task priority.
`set_task_expiration`	`eax: task,` `ebx: task_expiration`	`eax: task`	Sets the task expiration.
`set_task_duration`	`eax: task,` `ebx: task_duration`	`eax: task`	Sets the task duration.

Riferimenti bibliografici

[1] Andrew Morton, Jeffrey Liu, Insop Song (2007), EFFICIENT PRIORITY-QUEUE DATA STRUCTURE FOR HARDWARE IMPLEMENTATION.