1 Phase 1: Queue Queue, MiKABoO

(versione 0.1)

Ogni buon programma viene costruito a partire dalle strutture dati. Scopo della phase1 è di implementare la libreria di gestione delle code che verranno utilizzate dal kernel (phase2). In questo modo è possibile per gli studenti prendere confidenza con l'emulatore `uarm`, i metodi per generare il codice, gli strumenti di debug. Da un punto di vista organizzativo la phase1 consente di mettere a punto gli strumenti di collaborazione all'interno del gruppo, definire gli standard di documentazione e verificare di aver ben compreso gli elementi di valutazione del lavoro svolto.

La libreria di gestione delle code fa uso delle macro presenti nel file listx.c per la gestione delle liste bidirezionali. Queste macro sono le stesse usate all'interno del kernel Linux, infatti listx.h non è che un estratto del file `include/linux/list.h` presente nei sorgenti del kernel.

Tramite queste macro i gruppi sono chiamati a realizzare gli strumenti di gestione delle strutture dati relative a:

- Processi
- Thread
- Messaggi

rispettando le interfacce descritte in questo documento e fornite nel file `mikabooq.h`.

Il kernel non ha librerie di gestione dell'allocazione dinamica, non esistono quindi funzioni come *malloc* o *free*. L'allocazione della memoria per le strutture dati deve essere fornita dal kernel stesso.

Viene definito un vettore statico di elementi per ogni tipo di struttura C necessaria per mantenere le informazioni di un processo, di un thread o di un messaggio. Questi vettori sono dimensionati in modo tale che esistano elementi in numero sufficiente alla vita operativa del sistema (costanti fornite nel file `const.h`). Per ogni tipo di struttura in fase di inizializzazione gli elementi del vettore statico vengono inseriti in una lista (lista degli elementi liberi, o *lista libera*). Quando il sistema operativo ha necessità di un elemento per un nuovo processo, thread o messaggio chiama la funzione di "allocazione" che stacca il primo elemento dalla *lista libera* degli elementi del tipo richiesto e ne restituisce il puntatore. Per "riciclare" un elemento non più utile, questo viene reinserito nella *lista libera*.

1.1 Process Control Block

I processi in MiKABoO sono "contenitori" di thread. I thread dello stesso processo condividono le risorse. In MiKABoO l'astrazione dei processi viene mantenuta tramite la struttura dati Process Control Block (o in breve PCB).

```
struct pcb_t {
  struct pcb_t *p_parent; /* pointer to parent */
  struct list_head p_threads; /* list of threads */

  struct list_head p_children; /* list of children
                                  (hierarchy of processes) */
  struct list_head p_siblings; /* link the other siblings
                                  (children of p_parent) */
};
```

Questa struttura non contiene i campi che serviranno per la gestione delle risorse, campi che verranno aggiunti nello svolgimento delle prossime fasi del progetto.

I processi sono organizzati in una struttura ad albero. Ogni processo (diverso dal processo radice) ha un processo genitore `p_parent` e può avere processi figli. La lista dei processi figli ha come punto d'ingresso `p_children` e i fratelli vengono collegati tramite il campo `p_siblings`. (Il campo `p_siblings` serve anche per la gestione della *lista libera*).

Il campo `p_threads` consente di accedere alla lista dei thread del processo.

1.2 Funzioni di gestione dei PCB

- `struct pcb_t *proc_init(void);`

  Questa funzione inizializza la *lista libera* dei PCB e crea il processo radice. Al termine dell'esecuzione la *lista libera* deve contenere MAXPROC - 1 elementi e il puntatore al PCB del processo radice deve essere fornito come valore di ritorno. I campi del PCB radice devono avere valori coerenti.

- `struct pcb_t *proc_alloc(struct pcb_t *p_parent);`

  Alloca un elemento per un nuovo PCB e lo inserisce nella gerarchia dei processi come figlio del processo il cui PCB viene passato come parametro. Il valore di ritorno è il puntatore al PCB. Se non vi sono elementi disponibili nella *lista libera* viene restituito NULL. Se il parametro `p_parent` è NULL (caso di errore) la funzione non alloca alcun PCB e restituisce ugualmente NULL.

- `int proc_delete(struct pcb_t *oldproc);`

  Elimina il PCB passato come parametro. Un PCB si può eliminare solo se non ha nè figli nè thread. `proc_delete` restituisce 0 se l'operazione è stata completata, -1 in caso di errore. Eliminare un PCB significa toglierlo dalla gerarchia dei processi e reinserirlo nella *lista libera*.

- `struct pcb_t *proc_firstchild(struct pcb_t *proc);`

  Restituisce il puntatore al primo processo figlio (o NULL se il processo non ha figli).

- `struct tcb_t *proc_firstthread(struct pcb_t *proc);`

  Restituisce il puntatore al thread del processo figlio (o NULL se il processo non ha thread).

1.3 Thread Control Block

I thread sono le sequenze concorrenti di esecuzione che verranno gestite dallo scheduler del sistema operativo. La struttura dati che contiene le informazioni relative ad un thread è il *Thread Control Block* (in breve TCB).

```
struct tcb_t {
  struct pcb_t *t_pcb; /* pointer to the process */
  state_t t_s ; /* processor state */

  int t_status;

  struct tcb_t *t_wait4sender;
        /* expected sender (if t_status == T_STATUS_W4MSG),
             NULL means accept msg from anybody */
  struct list_head t_next;
        /* link the other elements of the list of threads
             in the same process */
  struct list_head t_sched;
        /* link the other elements on the same scheduling list */
  struct list_head t_msgq;
        /* list of pending messages for the current thread */
};
```

Come è già stato visto ogni thread appartiene ad un processo, il campo `t_pcb` collega ogni TCB al PCB del suo processo. Il campo `t_s` contiene lo stato del processore, la foto istantanea dello stato di esecuzione contenente il

valore di tutti i registri e utile allo scheduler per far ripartire l'esecuzione del thread esattamente dal punto nel quale era stata sospesa. Il campo `t_status` è una costante numerica che mantiene lo stato del thread che può assumere i valori:

- `T_STATUS_NONE` se il thread è stato allocato ma non ne è ancora iniziata l'esecuzione;
- `T_STATUS_READY` se il thread è schedulabile;
- `T_STATUS_W4MSG` se il thread è in stato di attesa (wait) per la ricezione di un messaggio.

Quando un thread è nello stato `T_STATUS_W4MSG` il campo `t_wait4sender` indica il TCB del thread del mittente atteso. Se `t_wait4sender` in questo caso è NULL il thread attende un messaggio proveniente da qualsiasi mittente.

N.B. i campi `t_s`, `t_status` e `t_wait4sender` non sono utili a phase1, verranno gestiti dal codice che scriverete per phase2. Sono stati inseriti per completezza, per far comprendere lo scopo della struttura. Altri campi per la gestione dei thread verranno inseriti nel TCB durante l'implementazione delle successive fasi del progetto.

Tutti i thread dello stesso processo sono collegati in una lista che ha come punto d'ingresso il campo `p_thread` del PCB e come collegamento il campo `t_next` del TCB. (`t_next` serve anche per la gestione della *lista libera*).

I thread possono di volta in volta entrare e uscire da code gestite dallo scheduler (per esempio dalla *ready queue*). Il campo `t_sched` serve appunto per poter inserire o togliere il processo da queste code.

Infine tutti i messaggi in attesa di essere ricevuti da un thread vengono mantenuti in una coda il cui punto d'ingresso è costituito dal campo `t_msgq`.

1.4 Funzioni di gestione dei TCB

- `void thread_init(void);`

  Crea la *lista libera* per i PCB (MAXTHREAD elementi).

- `struct tcb_t *thread_alloc(struct pcb_t *process);`

  Alloca un elemento per un nuovo thread estraendolo dalla lista degli elementi liberi e lo aggiunge ai thread del processo del quale il PCB è passato come parametro. Restituisce NULL in caso di errore e cioè se non vi sono elementi allocabili nella *lista libera* oppure se il puntatore al PCB passato come parametro è NULL.

- `int thread_free(struct tcb_t *oldthread);`

  Dealloca un TCB. L'operazione è consentita solo se la coda dei messaggi è vuota. Il valore di ritorno è 0 in caso di successo -1 altrimenti. La funzione deve anche provvedere a togliere l'elemento dalla lista dei thread del processo.

1.5 Funzioni di gestione delle code dello scheduler

- `void thread_enqueue(struct tcb_t *new, struct list_head *queue);`

  Quest funzione inserisce il TCB `new` nella coda che ha `queue` come punto di ingresso.

- `struct tcb_t *thread_qhead(struct list_head *queue);`

  Questa funzione restituisce il puntatore al PCB che costituisce la testa della coda che ha `queue` come punto di ingresso senza modificare la coda. La funzione restituisce NULL se la coda è vuota.

- `struct tcb_t *thread_dequeue(struct list_head *queue);`

  Questa funzione disaccoda un elemento dalla coda che ha `queue` come punto di ingresso. La funzione restituisce il puntatore all'elemento estratto dalla coda, NULL se la coda è vuota.

1.6 Code dei messaggi

In MiKABoO i "messaggi" hanno l'ampiezza di un indirizzo di memoria. Possono venir utilizzati per trasmettere o un indirizzo o un numero intero. Gli attori in grado di spedire e ricevere i messaggi sono i thread. Pertanto per individuare il mittente o il destinatario di un messaggio vengono utilizzati puntatori ai TCB del thread che spedisce o che deve ricevere il messaggio.

La coda dei messaggi in attesa di essere ricevuti da un thread viene gestita tramite la seguente struttura:

```
struct msg_t {
  struct tcb_t *m_sender; /* sender thread */
  uintptr_t m_value; /* payload of the message */

  struct list_head m_next; /* link the other elements of the
                              pending message queue */
};
```

Il valore del messaggio è `m_value`, mentre `m_sender` identifica il thread mittente. Il campo `m_next` collega fra loro i messaggi spediti allo stesso thread in una coda che ha come punto d'ingresso il campo `t_msgq` del TCB del destinatario.

1.7 Funzioni di gestione delle code di messaggi

- `void msgq_init(void);`

  Inizializza la *lista libera* per le strutture `struct msg_t` (di MAXMSG elementi).

- `int msgq_add(struct tcb_t *sender, struct tcb_t *destination, uintptr_t value);`

  Aggiunge un messaggio che ha come mittente `sender` e come valore `value` nella coda dei messaggi in attesa di essere ricevuti da `destination`. Se i parametri `sender` o `destination` hanno valore NULL o se la *lista libera* dei lessaggi è vuota, `msgq_add` restituisce il valore di errore -1. In caso di successo la funzione restituisce 0.

- `int msgq_get(struct tcb_t **sender, struct tcb_t *destination, uintptr_t *value);`

  Prende un messaggio dalla coda di quelli in attesa di essere ricevuti da `destination` e ne pone il valore nella variabile puntata da `value`. Il parametro `sender` consente di selezionare, se necessario, il mittente del messaggio. A seconda del valore di `sender` possono essere selezionati tre casi diversi:

  - `sender == NULL` restituisce il primo messaggio in coda qualsiasi ne sia il mittente. L'indirizzo del TCB del mittente non viene restituito;
  - `sender != NULL && *sender == NULL` restituisce il primo messaggio in coda qualsiasi ne sia il mittente. L'indirizzo del TCB del mittente viene memorizzato in `*sender`;
  - `sender != NULL && *sender != NULL` restituisce il primo messaggio in coda che ha `*sender` come mittente.

  Nel caso nessun messaggio nella coda soddisfi le condizioni richieste, `msgq_get` restituisce -1. In caso di successo il valore di ritorno è 0, il valore del messaggio vene assegnato a `*value` e l'elemento `struct msg_t` viene reinserito nella *lista libera* per poter essere riutilizzato.