RiKaya Operating System

Specifiche di Progetto

FASE 1

v.0.2

Anno Accademico 2018-2019
(da un documento di Marco di Felice)

RiKaya OS

- RiKaya: Evoluzione di Kaya O.S., a sua volta evoluzione di una lunga lista di S.O. proposti a scopo didattico (HOCA, TINA, ICARO, etc).
- RiKaya deve essere realizzato su architettura **uMPS2**
- Architettura basata su **sei** *livelli di astrazione*, sul modello del S.O. THE proposto da Dijkstra in un suo articolo del 1968 …

RiKaya OS

- Sistema Operativo in 6 **livelli** di astrazione.

Livello 6: *Shell interattiva*

Livello 5: *File-system*

Livello 4: *Livello di supporto*

Livello 3: *Kernel del S.O.*

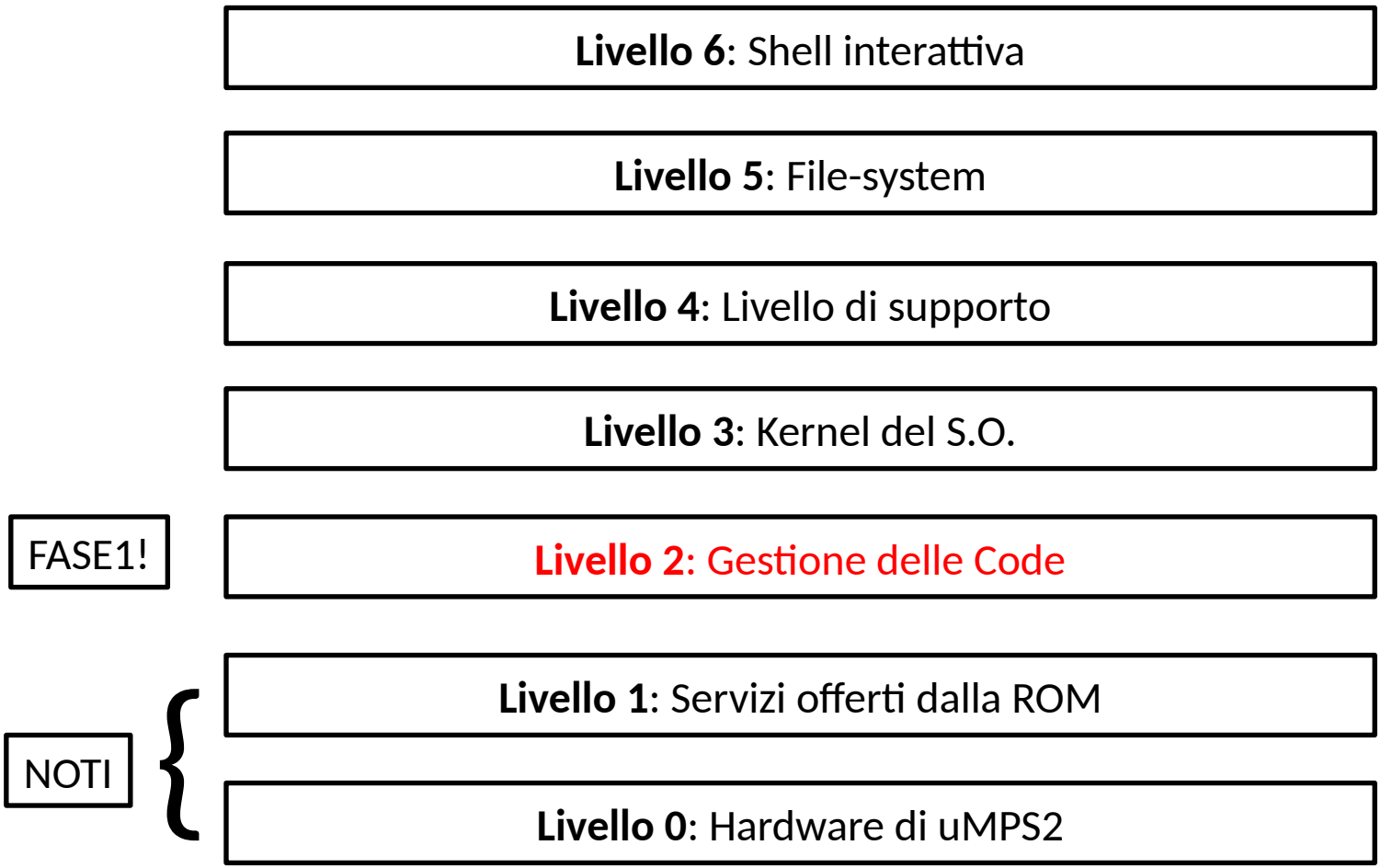
Livello 2: *Gestione delle Code*

Livello 1: *Servizi offerti dalla ROM*

Livello 0: *Hardware di uMPS2*

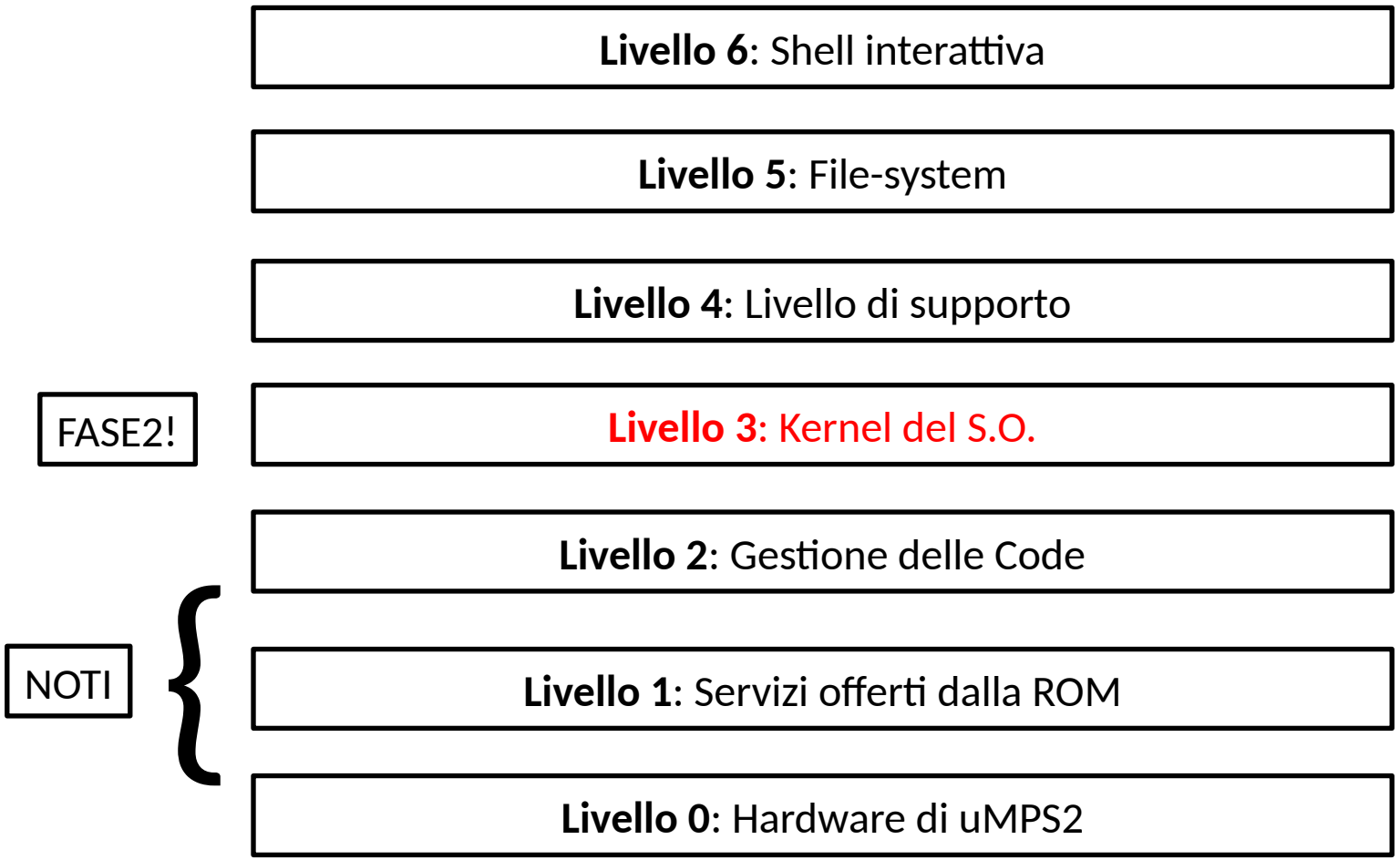
RiKaya OS

- Sistema Operativo in 6 **livelli** di astrazione.



RiKaya OS

- Sistema Operativo in 6 **livelli** di astrazione.



Livello 2 del S.O.

- Il livello 2 di RiKaya (Livello delle Code) fornisce l'implementazione delle **strutture dati** utilizzate dal livello sovrastante (**kernel**).
- Processi (livello 3)
- Process Control Block (**PCB**) livello 2.

```
typedef struct pcb_t {
    struct list_head p_next;
    struct pcb_t* p_parent;
    struct list_head p_child;
    struct list_head p_sib;
    state_t p_s;
    int priority;
    int p_semKey;
}
```

Livello 2 del S.O.

- Il gestore delle code implementa 4 funzionalita' relative ai PCB:
- **Allocazione e Deallocazione** dei PCB.
- Gestione delle **Code** dei PCB.
- Gestione dell'**Albero** dei PCB.
- Gestione di una **Active Semaphore List** (ASL), che gestisce la coda dei processi bloccati su un semaforo.

ASSUNZIONE: non più di MAXPROC processi concorrenti in RiKaya. MAXPROC = 20 (file const.h).

Allocazione dei PCB

- `pcbFree`: lista dei PCB che sono liberi o inutilizzati.
- `pcbfree_h`: elemento sentinella della lista pcbFree.
- `pcbFree_table[MAX_PROC]`: array di PCB con dimensione massima di MAX_PROC.

FUNZIONI da IMPLEMENTARE:

1. `void initPcbs()`

DESCRIZIONE: Inizializza la pcbFree in modo da contenere tutti gli elementi della pcbFree_table. Questo metodo deve essere chiamato una volta sola in fase di inizializzazione della struttura dati.

Allocazione dei PCB

FUNZIONI da IMPLEMENTARE:

2. `void freePcb(pcb_t * p)`

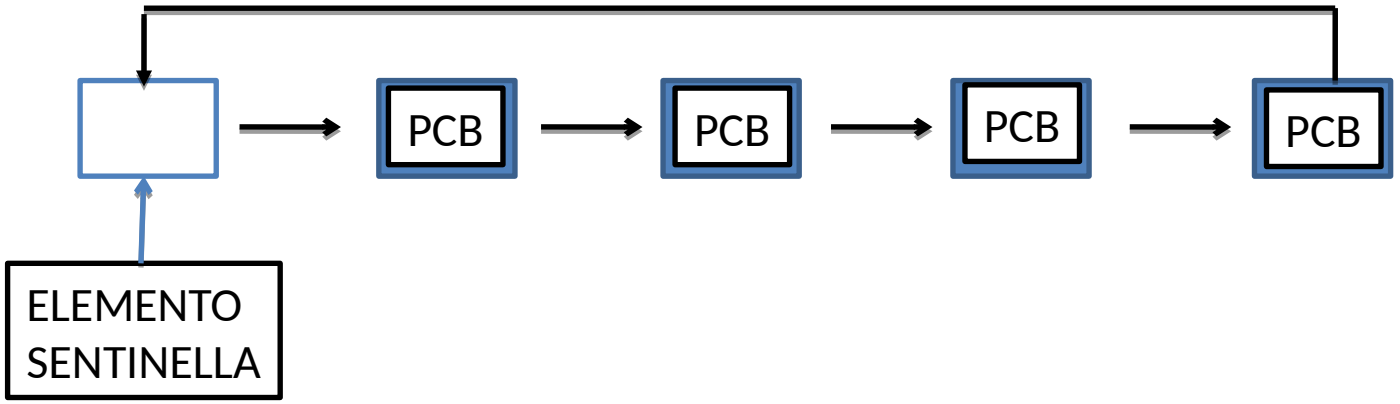
DESCRIZIONE: Inserisce il PCB puntato da p nella lista dei PCB liberi (pcbFree)

3. `pcb_t *allocPcb()`

DESCRIZIONE: Restituisce NULL se la pcbFree è vuota. Altrimenti rimuove un elemento dalla pcbFree, inizializza tutti i campi (NULL/0) e restituisce l'elemento rimosso.

Lista dei PCB

- I **PCB** possono essere organizzati in code, dette code di processi (es. coda dei processi attivi).
- Collegamento tra elementi: singolo o doppio
- Ciascuna lista è gestita tramite *`list_head`* (elemento sentinella – dummy).



Lista dei PCB

- **FUNZIONI DA IMPLEMENTARE:**

4. `pcb_t* mkEmptyProcQ(struct list_head* head)`

DESCRIZIONE: Inizializza la lista dei PCB, inizializzando l'elemento sentinella.

5. `int emptyProcQ(struct list_head* head)`

DESCRIZIONE: Restituisce TRUE se la lista puntata da head è vuota, FALSE altrimenti.

Lista dei PCB

- **FUNZIONI DA IMPLEMENTARE**:

6. `void insertProcQ(struct list_head* head, pcb* p)`

DESCRIZIONE: inserisce l'elemento puntato da p nella coda dei processi puntata da head. L'inserimento deve avvenire tenendo conto della priorita' di ciascun pcb (campo p->priority). La coda dei processi deve essere ordinata in base alla priorita' dei PCB, in ordine decrescente (i.e. l'elemento di testa è l'elemento con la priorita' più alta).

7. `pcb_t headProcQ(struct list_head* head)`

DESCRIZIONE: Restituisce l'elemento di testa della coda dei processi da head, SENZA RIMUOVERLO. Ritorna NULL se la coda non ha elementi.

Lista dei PCB

- **FUNZIONI DA IMPLEMENTARE:**

8. `pcb_t* removeProcQ(struct list_head* head)`

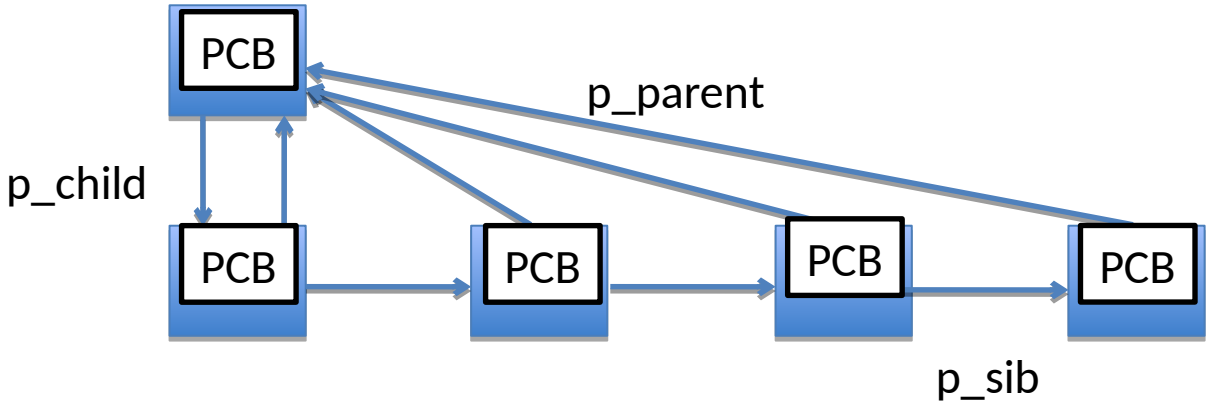
DESCRIZIONE: rimuove il primo elemento dalla coda dei processi puntata da head. Ritorna NULL se la coda è vuota. Altrimenti ritorna il puntatore all'elemento rimosso dalla lista.

9. `pcb_t* outProcQ(struct list_head* head, pcb_t *p)`

DESCRIZIONE: Rimuove il PCB puntato da p dalla coda dei processi puntata da head. Se p non è presente nella coda, restituisce NULL. (NOTA: p può trovarsi in una posizione arbitraria della coda).

Alberi di PCB

- In aggiunta alla possibilita' di partecipare ad una coda di processo, i PCB possono essere organizzati in **alberi** di processi .
- Ogni genitore contiene un list_head (p_child) che punta alla lista dei figli.
- Ogni figlio ha un puntatore al padre (p_parent) ed un list_head che punta alla lista dei fratelli.



Lista dei PCB

- **FUNZIONI DA IMPLEMENTARE:**

10. `int emptyChild(pcb_t *p)`

DESCRIZIONE: restituisce TRUE se il PCB puntato da p non ha figli, restituisce FALSE altrimenti.

11. `void insertChild(pcb_t *prnt,pcb_t *p)`

DESCRIZIONE: Inserisce il PCB puntato da p come figlio del PCB puntato da prnt.

12. `pcb_t* removeChild(pcb_t *p)`

DESCRIZIONE. Rimuove il primo figlio del PCB puntato da p. Se p non ha figli, restituisce NULL.

Lista dei PCB

- **FUNZIONI DA IMPLEMENTARE:**

13. `pcb_t *outChild(pcb_t* p)`

DESCRIZIONE: Rimuove il PCB puntato da p dalla lista dei figli del padre. Se il PCB puntato da p non ha un padre, restituisce NULL. Altrimenti restituisce l'elemento rimosso (cioè p). A differenza della removeChild, p può trovarsi in una posizione arbitraria (ossia non è necessariamente il primo figlio del padre).

Semafori

- In RiKaya, l'accesso alle risorse condivise avviene attraverso l'utilizzo di **semafori**.
- Ad ogni semaforo è associato un **descrittore (SEMD)** con la struttura seguente:

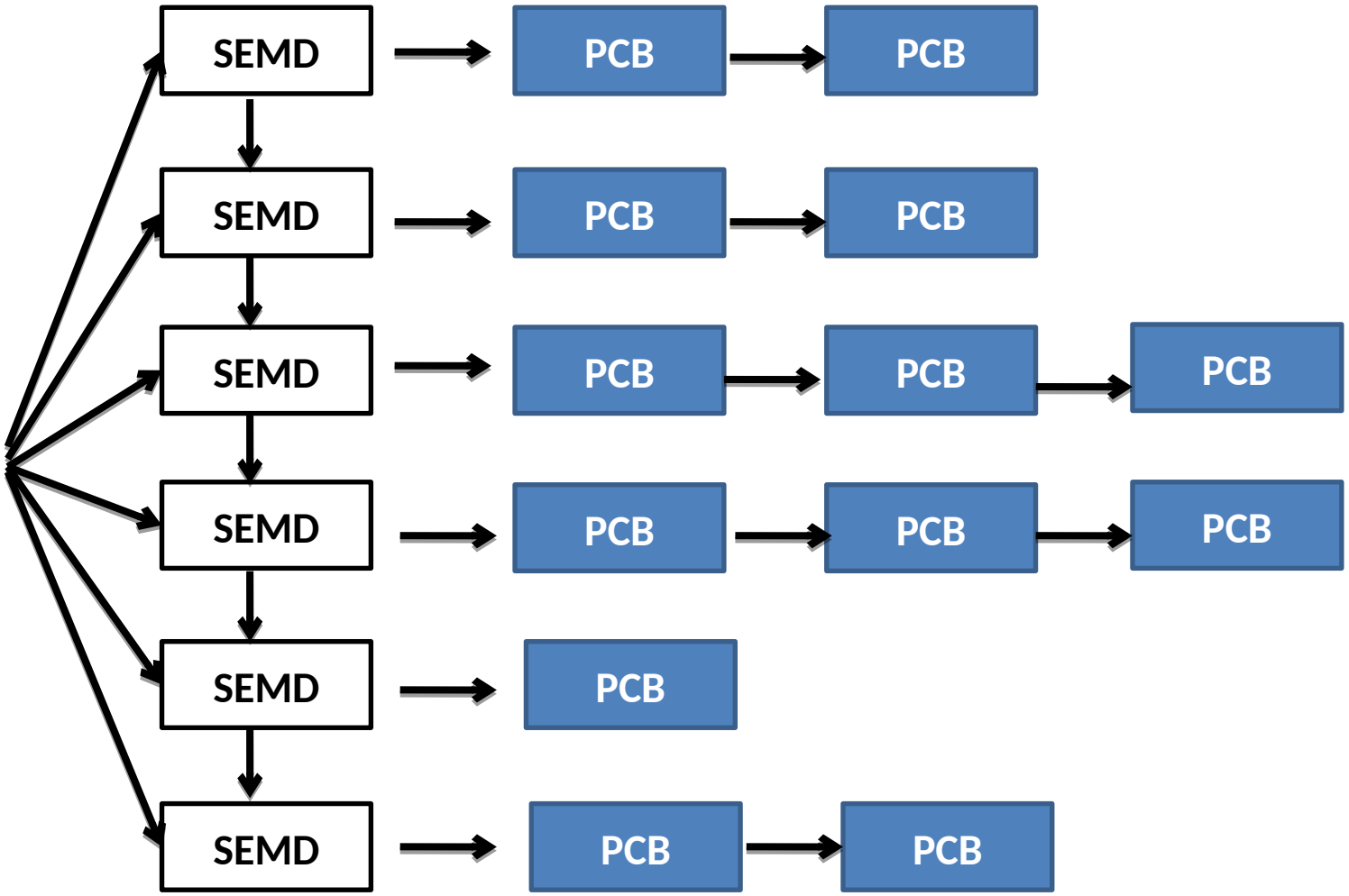
```
typedef struct semd_t {
        struct list_head s_next;
        int   *s_key;
        struct list_head s_procQ;
} semd_t;
```

Active Semaphore List

- `semd_table[MAX_PROC]`: array di SEMD con dimensione massima di MAX_PROC.
- `semdFree`: Lista dei SEMD liberi o inutilizzati.
- `semdFree_h`: elemento sentinella della lista semdFree.
- E' necessario gestire la lista dei SEMD attivi (*Active Semaphore List* – **ASL**)
- `semd_h`: elemento sentinella della lista ASL.

Active Semaphore List

semd_t*
getSemd
(int key)



Gestione della ASL

- **FUNZIONI DA IMPLEMENTARE:**

14. `semd_t* getSemd(int *key)`

DESCRIZIONE: restituisce il puntatore al SEMD nella ASL la cui chiave è pari a key. Se non esiste un elemento nella ASL con chiave eguale a key, viene restituito NULL.

15. `int insertBlocked(int *key,pcb_t *p)`

DESCRIZIONE: Viene inserito il PCB puntato da p nella coda dei processi bloccati associata al SEMD con chiave key. Se il semaforo corrispondente non è presente nella ASL, alloca un nuovo SEMD dalla lista di quelli liberi (semdFree) e lo inserisce nella ASL, settando I campi in maniera opportuna (i.e. key e s_procQ). Se non è possibile allocare un nuovo SEMD perché la lista di quelli liberi è vuota, restituisce TRUE. In tutti gli altri casi, restituisce FALSE.

Gestione della ASL

- **FUNZIONI DA IMPLEMENTARE:**

16. `pcb_t* removeBlocked(int *key)`

DESCRIZIONE: Ritorna il primo PCB dalla coda dei processi bloccati (s_ProcQ) associata al SEMD della ASL con chiave key. Se tale descrittore non esiste nella ASL, restituisce NULL. Altrimenti, restituisce l'elemento rimosso. Se la coda dei processi bloccati per il semaforo diventa vuota, rimuove il descrittore corrispondente dalla ASL e lo inserisce nella coda dei descrittori liberi (semdFree).

Gestione della ASL

- **FUNZIONI DA IMPLEMENTARE:**

17. `pcb_t* outBlocked(pcb_t *p)`

DESCRIZIONE: Rimuove il PCB puntato da p dalla coda del semaforo su cui è bloccato (indicato da p->p_semKey). Se il PCB non compare in tale coda, allora restituisce NULL (condizione di errore). Altrimenti, restituisce p.

18. `pcb_t* headBlocked(int *key)`

DESCRIZIONE: Restituisce (senza rimuovere) il puntatore al PCB che si trova in testa alla coda dei processi associata al SEMD con chiave key. Ritorna NULL se il SEMD non compare nella ASL oppure se compare ma la sua coda dei processi è vuota.

Gestione della ASL

- **FUNZIONI DA IMPLEMENTARE:**

19. `void outChildBlocked(pcb_t *p)`

DESCRIZIONE: Rimuove il PCB puntato da p dalla coda del semaforo su cui è bloccato (indicato da p->p_semKey). Inoltre, elimina tutti i processi dell'albero radicato in p (ossia tutti i processi che hanno come avo p) dalle eventuali code dei semafori su cui sono bloccati.

20. `void initASL()`

DESCRIZIONE: Inizializza la lista dei semdFree in modo da contenere tutti gli elementi della semdTable. Questo metodo viene invocato una volta sola durante l'inizializzazione della struttura dati.

Consigli

- Non esiste un metodo univoco per l'implementazione delle strutture dati di Fase1.
- **Suggerimento**: Creare due moduli separati, uno per la gestione dei PCB ed uno per la gestione dei SEMD.
- Usare ove possibile metodi/variabili HIDDEN.
- Dichiarazione di pcb_t e semd_t in types.h

Compilazione

```
mipsel-linux-gnu-gcc -ansi -mips1 -mfp32
-I/usr/local/include/umps2 -c ???.c

mipsel-linux-gnu-ld -o kernel ???.o
/usr/local/lib/umps2/crtso.o
/usr/local/lib/umps2/libumps.o -nostdlib -T /
usr/local/share/umps2/umpscore.ldscript
```

ESECUZIONE

- `umps2-elf2umps -k kernel`
- LANCIARE UMPS2 E SETTARE IN MANIERA OPPORTUNA IL **KERNEL.CORE** DELLA MACCHINA VIRTUALE
- CONTROLLARE LO STATO DI AVANZAMENTO DEL TEST SUL TERMINALE …
- **SYSTEM HALTED:** OK, TEST COMPLETATO
- **KERNEL PANIC:** TERMINAZIONE FORZATA … (QUALCHE PROBLEMA …)

CONSEGNA

- La deadline di consegna è fissata per il giorno: **Domenica 24 Febbraio 2019, ore 23.59**
- CONSEGNARE IL PROPRIO PROGETTO (un unico file .tar.gz) NELLA **CARTELLA DI CONSEGNA** ASSOCIATA AL PROPRIO GRUPPO.
- CONSEGNARE ENTRO LA **DEADLINE** FISSATA.
- **VERIFICARE CHE L'ARCHIVIO .TAR.GZ NON SIA ROVINATO.**

CONSEGNA

- Cosa consegnare:
  - Sorgenti del progetto (TUTTI)
  - Makefile per la compilazione
  - Documentazione (scelte progettuali, eventuali modifiche effettuate nelle specifiche)
  - File AUTHORS, README (eventuale)

- Inserire **commenti** nel codice per favorire la leggibilita' e la correzione …
  - **PROGETTI non COMMENTATI NON SARANNO VALUTATI.**

RiKaya Operating System

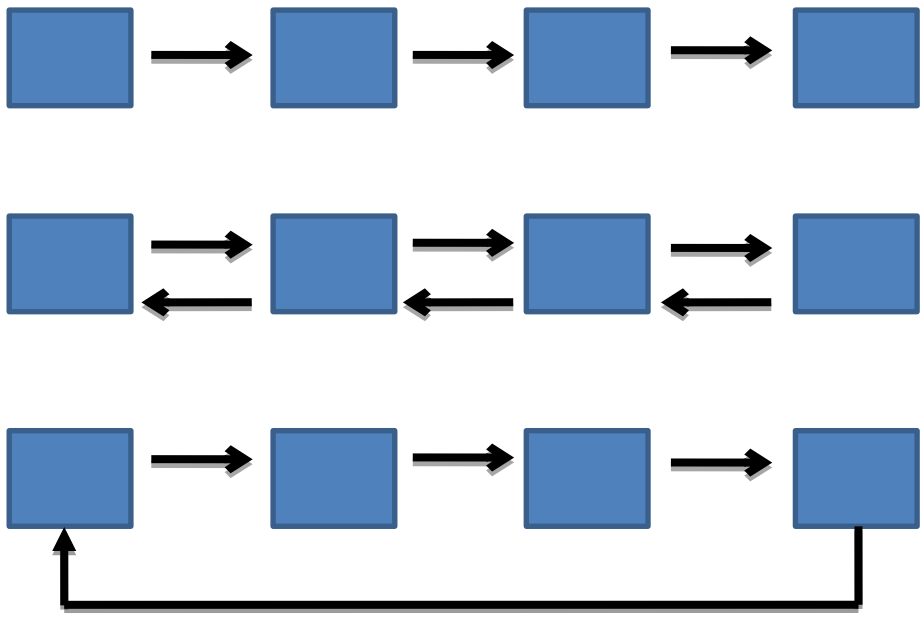
Compendio:

Le liste del Kernel di Linux

Anno Accademico 2018-2019

Lista di Elementi

- Una **lista** è una struttura dati che contiene un certo numero di elementi concatenati tra loro.
- Diverse modalita' di **collegamento** tra elementi:



COLLEGAMENTO SINGOLO

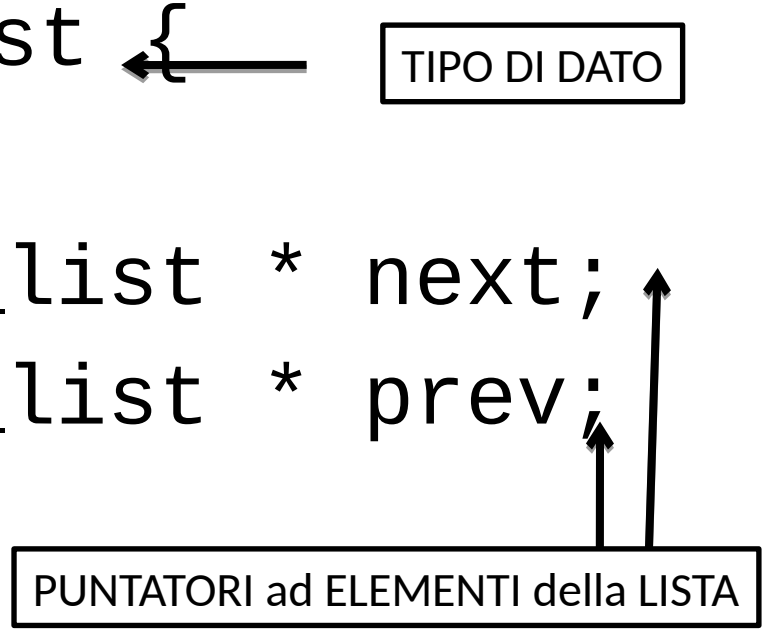
COLLEGAMENTO DOPPIO

COLLEGAMENTO CIRCOLARE

Definizione classica

- Una **lista** è definita insieme al tipo di dato gestito:

- ```
  struct item_list {
      int item;
      struct item_list * next;
      struct item_list * prev;
  }
  ```


  ```

Definizione classica

- Una **lista** è definita insieme al tipo di dato gestito:

- ```
  struct item_list {
      int item;
      struct item_list * next;
      struct item_list * prev;
  }
  ```

PROBLEMA: Dipendenza esplicita dal tipo di dato !!
  ```

Liste del Linux-Kernel

- Il **kernel** di Linux fornisce una libreria per la manipolazione di **liste generiche**, indipendenti dal tipo di dato gestito (**type oblivious**).
- Implementazione di lista doppia concatenata.
- Caratteristiche:
  - **Codice riutilizzabile** per qualsiasi dato/struttura
  - Utilizzo di un **elemento sentinella** (dummy).
    - *Non rappresenta un elemento della lista*
    - *Serve per concatenare il primo e l'ultimo elemento.*

Liste del Linux-Kernel

- L'elemento chiave delle liste è il **concatenatore** (struct list_head), che unisce l'elemento attuale con quello successivo.
- Definito come una **coppia di puntatori** all'elemento precedente e successivo della lista.

```
struct list_head {
      struct list_head* next;
      struct list_head* prev;
   }
```

NON CONTIENE RIFERIMENTO DIRETTO AI DATI!!

Liste del Linux-Kernel

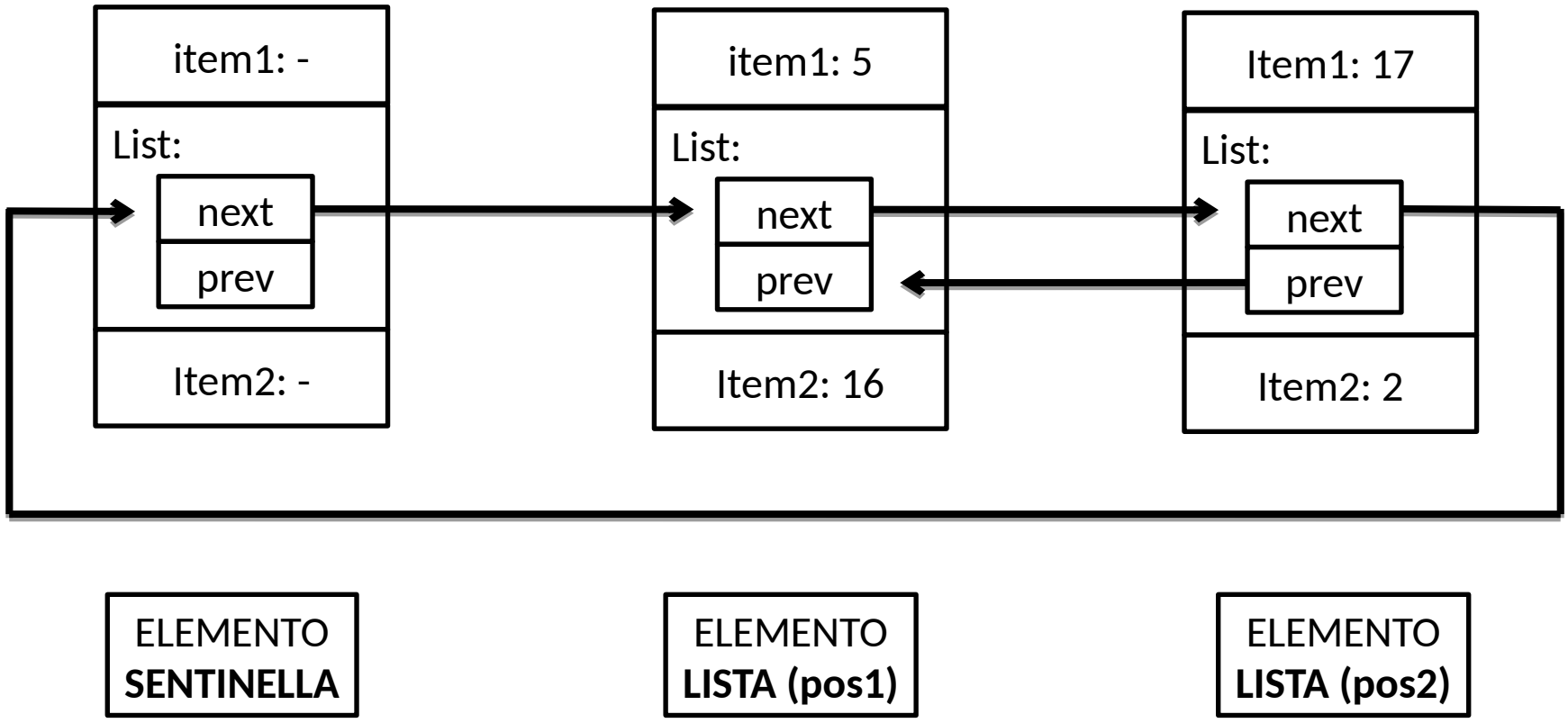
- Il concatenatore deve essere inserito all'interno della lista per collegarsi agli elementi successivi.

```
struct mylist {
   int item1;
     struct list_head list;
     int item2;
  }
```

Il concatenatore può essere inserito ovunque …

Il concatenatore può essere chiamato in qualsiasi modo.

Liste del Linux-Kernel



API per la gestione Liste

- Esistono un insieme di macro (include/linux/list.h) per la manipolazione/creazione delle liste ...

1. *Inizializzazione di una lista vuota*
2. *Aggiunta di un elemento in diverse posizioni*
3. *Cancellazione di un elemento da una lista*
4. *Controllo di lista vuota*
5. *Scorrimento di una lista*

Inizializzazione di Liste

- Inizializzazione dell'elemento sentinella in modo da far puntare i campi prev e next alla struttura list_head che li contiene.
- Funzione inline `INIT_LIST_HEAD (&list)`

  list->next=list;

  list->prev=list;
- Altre macro per inizializzazione di liste:
  - `LIST_HEAD_INIT(list)`
  - `LIST_HEAD(#nome variabile)`

Verifica di Lista Vuota

- Funzione inline: list_empty()

```
int list_empty(struct list_head *head)
```

- *head è il list_head dell'elemento sentinella
- Restituisce TRUE se la lista è vuota, FALSE altrimenti … Come? Verificando se i campi next e prev della sentinella puntano alla sentinella stessa…

```
return (head->next == head->prev)
```

Aggiunta di un elemento

- Funzione inline: list_add()

```
void list_add(struct list_head *new,
struct list_head *head)
```

- *new è il puntatore al list_head del dato che si vuole inserire nella lista
- *head è il puntatore all'elemento sentinella
- L'elemento puntato da new è inserito IN TESTA:

```
head->next->prev = new;
new->next  = head->next;
new->prev  = head;
head->next = new;
```

Aggiunta di un elemento

- Funzione inline: list_add_tail(): Aggiunge un elemento in coda alla lista

```
void list_add_tail(struct list_head
*new, struct list_head *head)
```

- Funzione inline: __list_add_tail(): Aggiunge un elemento in posizione qualsiasi

```
void __list_add(struct list_head *new,
   struct list_head *prev,
   struct list_head *next)
```

Rimozione di un elemento

- Funzione inline: list_del(): Rimuove un elemento dalla lista

```
void list_del(struct list_head *entry)
```

- Il puntatore *entry punta al list_head dell'elemento che si desidera eliminare.
- Non viene deallocata la struttura dati puntata da *entry, ma viene solo staccato il list_head dalla lista:

```
next->prev  = prev;
prev->next  = next;
```

Accesso agli elementi

- Macro: #container_of(ptr, type, member): Estrae una struttura dati dal contenitore.

```
#define container_of(ptr, type, member)
```

- *ptr punta al list_head della struttura dati di cui si vuole ottenere un puntatore.
- type è il tipo di dato della struttura dati contenente il list_head.
- member è il nome della variabile list_head all'interno della struttura dati.
- Ritorna il puntatore alla struttura dati contenente *ptr.

Scorrimento di lista

- Macro: #list_for_each(): Consente di scorrere il contenuto di una lista

```
#define list_for_each(pos, head)
```

- *pos è una variabile di tipo list_head*.
- *head è il puntatore all'elemento sentinella.
- Scorre i list_head con il ciclo seguente:

```
for (pos=(head->next; pos!=(head);
   pos=pos->next)
```

Scorrimento di lista

- Macro: #list_for_each_entry(): Consente di scorrere il contenuto di una lista

```
#define list_for_each_entry(pos, head, member)
```

- *pos è una variabile di tipo puntatore all'elemento.
- *head è il puntatore all'elemento sentinella.
- member è il nome del list_head
- Scorre la lista ed assegna il puntatore all'elemento corrente a pos