

# YaeOS

Progetto di Sistemi Operativi  
Anno Accademico 2017/18

**Professore:** Renzo Davoli

## **Gruppo 13:**

Lorenzo Poluzzi

Lorenzo Turini

Melania Ghelli

Daniel Veroli

## **Introduzione**

Il lavoro che abbiamo svolto fa parte di un progetto più grande il cui scopo è la realizzazione di un intero Sistema Operativo. L'intero lavoro si articola in un certo numero di livelli e relativi moduli da implementare; in particolare durante questa prima fase ci siamo occupati del secondo, inerente la gestione delle code. Questo poggia sui due livelli inferiori: quello hardware e i servizi offerti dalla ROM, e implementa la gestione delle code di strutture dati che verranno poi utilizzate dal livello superiore: il Kernel.

Per svolgere questo compito ci siamo serviti dell'emulatore hardware uARM, basato su architettura ARM7TDMI, sviluppato da Marco Melletti a scopo didattico. Questo ha fornito gli strumenti necessari durante la fase di test. Abbiamo, inoltre, fatto largo uso di software *git* il quale si è dimostrato uno strumento molto utile e professionale per l'organizzazione e lo sviluppo del progetto.

## **Gestione delle code**

Questa prima fase richiedeva di implementare due diversi moduli: il ***pcb.c*** e ***asht.c***. Il primo fornisce i metodi per l'allocazione e la deallocazione della memoria, gestione delle code e degli alberi di Process Control Block (PCB). Il secondo, invece, permette la gestione dei semafori attivi tramite l'uso di funzioni hash e relative hash table.

## **Process Control Block (PCB.c)**

Per questa parte del lavoro ci siamo attenuti alle specifiche, non riscontrando particolari difficoltà. Il lavoro è stato organizzato attorno alle funzioni di gestione delle code (*insertProcQ*, *removeProcQ*, *headProcQ*, *outProcQ*, *forallProcQ*), le quali vengono poi richiamate dagli altri metodi.

Per la realizzazione delle code con priorità abbiamo scelto di utilizzare delle liste mono direzionali, in modo tale da sfruttare al meglio i campi della struttura **pcb\_t**. Nello specifico il campo **p\_next** è servito a collegare fra di loro gli elementi di tale lista, mentre abbiamo usato **p\_priority** come termine di confronto per ordinare correttamente gli elementi della coda.

Anche per la parte relativa agli alberi di PCB ci siamo limitati a manipolare i campi della struttura **pcb\_t**: **p\_parent**, **p\_child** e **p\_sib**.

## Active Semaphore Hash Table (ASHT.c)

In Yae OS, l'accesso alle risorse condivise avviene tramite l'utilizzo di semafori. Ad ogni semaforo è associato un descrittore noto come SEMD, il quale, fra i vari campi, contiene un intero il quale ha una doppia valenza: il valore vero e proprio rispecchia lo stato interno del semaforo, mentre il puntato all'intero viene utilizzato come identificativo univoco per fare riferimento ad ogni semaforo. I descrittori dei semafori vengono inizialmente allocati all'interno di un array di SEMD (**semd\_table[MAXSEMD]**) di dimensione massima MAXSEMD. Per la gestione dei semafori attivi viene utilizzata una lista di SEMD e un relativo puntatore alla testa **semdFree\_h** utilizzato per accedervi. I SEMD attivi vengono gestiti tramite una Hash Table (**semdhash[ASHDSIZE]**), all'interno della quale ogni elemento punta alla lista di collisione per il valore di hash corrispondente all'indice. Per la scelta della funzione di hash, un problema non irrilevante è stato trovare un buon algoritmo che fosse in grado di minimizzare il numero di collisioni delle chiavi sui vari indici della tabella, e quindi garantire una distribuzione equa. A tale proposito abbiamo sviluppato un programma di test per analizzare le statistiche relative a diversi algoritmi (numero di collisioni, distribuzione delle chiavi). L'algoritmo da noi scelto fornisce una distribuzione statistica molto equilibrata, e lavora con interi a 32 bit, operazioni bitwise e modulari. Si basa inoltre su un particolare numero (**0x45d9f3b**) calcolato usando uno speciale programma multi-thread di test, il quale calcola l'“avalanche effect”, ovvero il numero di bit che variano nell'output in risposta a un singolo bit cambiato in input. Qui sotto è riportato l'output relativo al nostro test dell'algoritmo, con 20 chiavi da attribuire a 8 indici.

Value-0: 6422076	Hash(value): -608563529	Unhashed(value): 6422076	HashModule 8: 8
Value-1: 6422080	Hash(value): 514627178	Unhashed(value): 6422080	HashModule 8: 2
Value-2: 6422096	Hash(value): 1130633304	Unhashed(value): 6422096	HashModule 8: 0
Value-3: 6422088	Hash(value): -1603688978	Unhashed(value): 6422088	HashModule 8: 6
Value-4: 6422092	Hash(value): 1370021193	Unhashed(value): 6422092	HashModule 8: 4
Value-5: 6422096	Hash(value): 1848088825	Unhashed(value): 6422096	HashModule 8: 5
Value-6: 6422100	Hash(value): -1095000209	Unhashed(value): 6422100	HashModule 8: 7
Value-7: 6422104	Hash(value): -1860441121	Unhashed(value): 6422104	HashModule 8: 3
Value-8: 6422100	Hash(value): 230113022	Unhashed(value): 6422100	HashModule 8: 6
Value-9: 6422112	Hash(value): 1612025519	Unhashed(value): 6422112	HashModule 8: 7
Value-10: 6422116	Hash(value): 150024573	Unhashed(value): 6422116	HashModule 8: 5
Value-11: 6422120	Hash(value): 1797379230	Unhashed(value): 6422120	HashModule 8: 6
Value-12: 6422124	Hash(value): 798188207	Unhashed(value): 6422124	HashModule 8: 1
Value-13: 6422128	Hash(value): -13210192	Unhashed(value): 6422128	HashModule 8: 0
Value-14: 6422132	Hash(value): 180711244	Unhashed(value): 6422132	HashModule 8: 4
Value-15: 6422136	Hash(value): 31074275	Unhashed(value): 6422136	HashModule 8: 3
Value-16: 6422140	Hash(value): 1937351298	Unhashed(value): 6422140	HashModule 8: 7
Value-17: 6422144	Hash(value): 169149210	Unhashed(value): 6422144	HashModule 8: 2
Value-18: 6422148	Hash(value): -656798615	Unhashed(value): 6422148	HashModule 8: 1
Value-19: 6422152	Hash(value): 642139709	Unhashed(value): 6422152	HashModule 8: 5

```

Statistiche su 20 elementi:

////////HASH FUNCTION////////
SenKey: 0      Collision Number: 2
                -Chiave: 6422034
                Chiave: 6422129
SenKey: 1      == Collision Number: 3
                -Chiave: 6422042
                -Chiave: 6422124
                Chiave: 6422148
SenKey: 2      == Collision Number: 3
                -Chiave: 6422030
                -Chiave: 6422140
                Chiave: 6422144
SenKey: 3      == Collision Number: 1
                -Chiave: 6422136
SenKey: 4      Collision Number: 1
                -Chiave: 6422132
SenKey: 5      == Collision Number: 4
                -Chiave: 6422076
                -Chiave: 6422096
                -Chiave: 6422116
                Chiave: 6422152
SenKey: 6      == Collision Number: 3
                -Chiave: 6422038
                -Chiave: 6422100
                Chiave: 6422120
SenKey: 7      == Collision Number: 3
                -Chiave: 6422108
                -Chiave: 6422104
                Chiave: 6422112

```

```

-----HASH TABLE-----
Chiave: 6422030      SenKey: 0
Chiave: 6422120      SenKey: 0
-----
Chiave: 6422052      SenKey: 1
Chiave: 6422174      SenKey: 1
Chiave: 6422140      SenKey: 1
-----
Chiave: 6422030      SenKey: 2
Chiave: 6422140      SenKey: 2
Chiave: 6422144      SenKey: 2
-----
Chiave: 6422156      SenKey: 3
-----
Chiave: 6422132      SenKey: 4
-----
Chiave: 6422076      SenKey: 5
Chiave: 6422096      SenKey: 5
Chiave: 6422116      SenKey: 5
Chiave: 6422152      SenKey: 5
-----
Chiave: 6422088      SenKey: 6
Chiave: 6422108      SenKey: 6
Chiave: 6422120      SenKey: 6
-----
Chiave: 6422100      SenKey: 7
Chiave: 6422104      SenKey: 7
Chiave: 6422112      SenKey: 7
-----

```

## Organizzazione del codice

I file sono stati suddivisi in due directory, `h` e `phase1`, le quali contengono rispettivamente gli header da includere e i file sorgenti. Questi ultimi sono i file delle librerie `pcb.c` e `asht.c` ed il file `test-pcb.c`, necessario per testare le funzionalità dei primi due.