YaeOS

Progetto di Sistemi Operativi

Anno Accademico 2017/18

Professore: Renzo Davoli

Gruppo 13:

Lorenzo Poluzzi

Lorenzo Turrini

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)

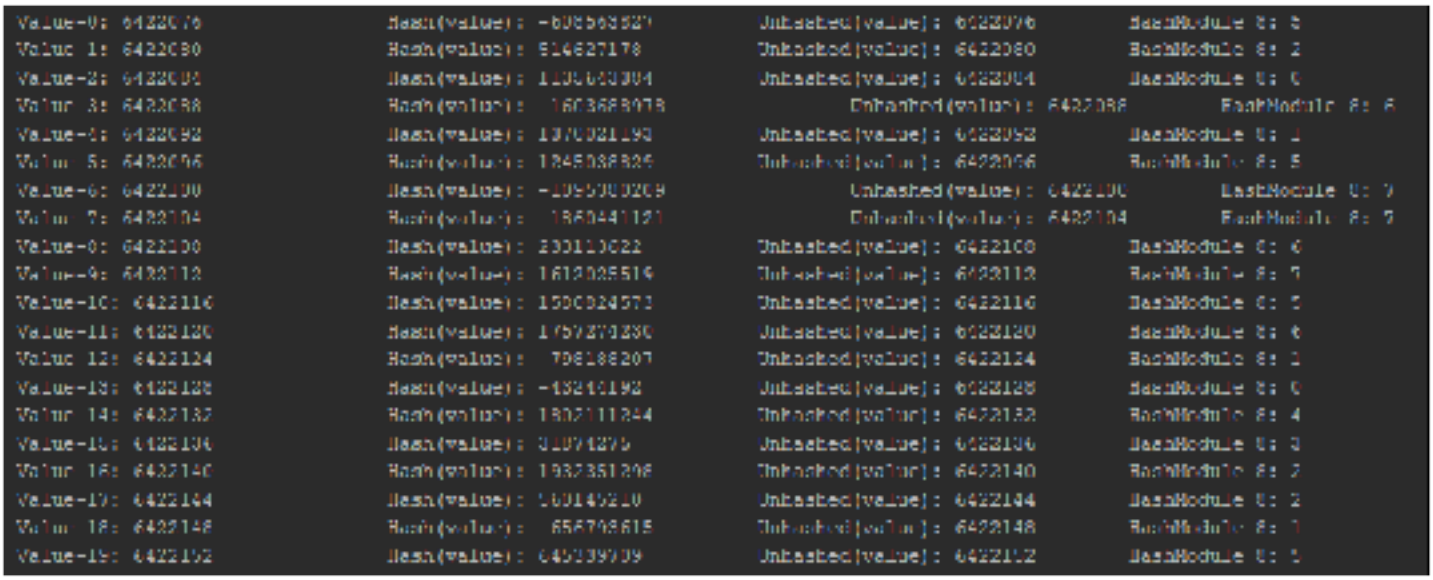
Per questa parte del lavoro ci siamo attenuti alle specifiche, non riscontrando particolari difficoltà. Il lavoro è stato organizzato attorno alle funzioni di gestione delle code (*insertProcQ, removeProcQ, headProcQ, outProcQ, forallProcQ*), le quali vengono poi richiamate dagli altri metodi.

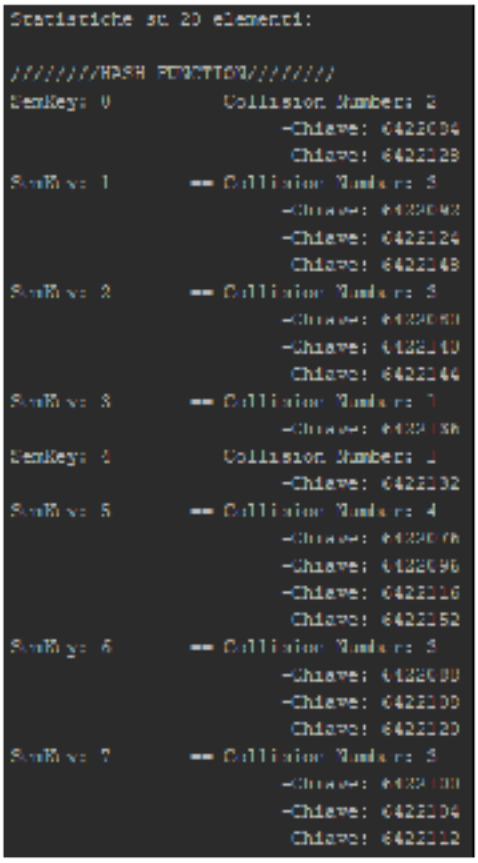
Per la realizzazione delle code con priorità abbiamo scelto di utilizzare delle liste monodirezionali, in modo tale da sfruttare al meglio i campi della struttura ***pcb_t***. Nello specifico il campo *p_next* è servito a collegare fra di loro gli elementi di tale lista, mentre abbiamo usato *p_priority*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

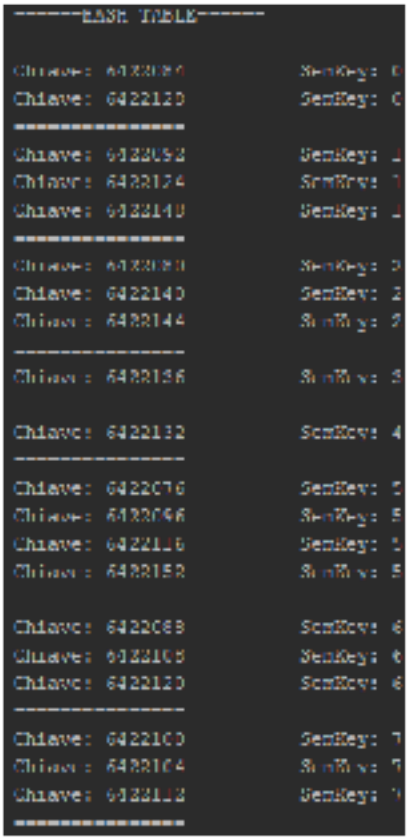
Anche per la parte relativa agli alberi di PCB ci siamo limitati a manipolare i campi della struttura ***pcb_t***: *p_parent*, *p_child* e *p_sib*.

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 chiave sui vari indici della tabella, e quindi garantirne 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.







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.