CheapOS

Progetto di Sistemi Operativi A.A. 2003/2004

Alessandro Fruchi
Tommaso Pignata

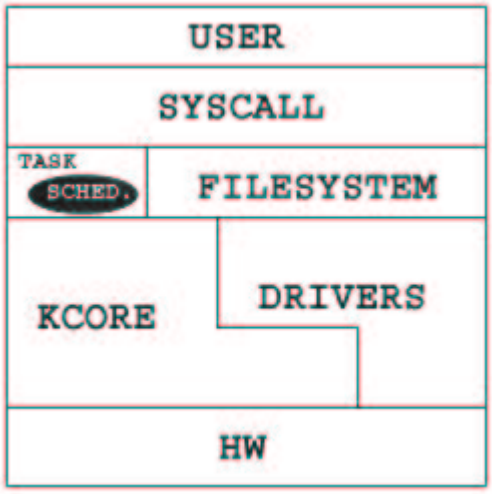
Manuale di sviluppo per cheapOS

Indice

Capitolo 1

Introduzione e stratificazione di cheapOS

Il codice di cheapOS e' organizzato in maniera stratificata per semplificarne la gestione,le modifiche ed il debug.
Possiamo schematizzarlo con la figura seguente:



KernelCore e' lo strato piu' profondo che, appoggiandosi direttamante all'Hw, fornisce una base per tutti gli altri.
KCore fornisce tuttavia funzioni anche molto potenti che suddividiamo in questi gruppi:

kconsole	--> Stampa a Video (interna) per il Debug
mem	--> Gestione della Memoria
cpu	--> Gestione IRQ ISR
task	--> Gestione dei task
time	--> Gestione della Temporizzazzione (clock)

Driver sono i driver dei dispositivi non 'essenziali' attualmente si appoggiano al KernelCore, ma eventualmente possono accedere direttamante all'Hw.
(Purtroppo non e' stato possibile per motivi di tempo creare un interfaccia generica comune a tutti i Driver!!)

Attualmente cheapOS supporta i seguenti dispositivi:

Tastiera
VideoVGA(modo testo)*
Floppy(singolo Drive)

*La gestione del VideoVGA e' indipendente dalla Kconsole nonche' molto piu' avanzata!

TaskMan (+_Scheduler_) e' la gestione avanzata dei task che appoggiandosi a quella primitiva del KernelCore implementa Multitasking, creazione processi per Fork ed Exec, distruzione di

processi su richiesta o perche' hanno eseguito operazioni non consentite**

**Attenzione: purtroppo la protezione non e' totale! Se il Task fa' qualcosa di davvero strano potrebbe crashare tutto cheapOS!

_FS_ sono i vari filesystem supportati da cheapOS (detti Volumi). L'unico modo che hanno i Task per comunicare con l'esterno ovvero accedere alle periferiche e' attraverso un apposito filesystem!

Attualmente i filesystem (volumi) supportati sono:

filesystem	nome volume	descrizione
fat12	FAT:	accesso al floppy (in sola lettura)
kernel	KER:	informazioni dal kernel (tipo /proc)
console	CON:	accesso alle console

Per implementare un altro filesystem o modificarne uno bisogna attenersi alle specifiche di un interfaccia comune a tutti la _FSI_ !

SYSCALL e' la gestione generica delle syscall (funzioni del kernel chiamate dei task); sono attualmente raggruppate in 3 blocchi a seconda delle funzioni.
In teoria non c'e' bisogno di aggiungerne altre!

Gruppo	A cosa servono
System	riguardanti il sistema in se' (ora solo shutdown)
Task	per la gestione di un task Fork Exit GetPid etc etc...
Filesystem	per l'accesso ai file Open Read Remv.....

Capitolo 2

KernelCore

Adesso parleremo passo passo delle funzioni che il KernelCore mette a disposizione specificandone le carateristiche, l'uso e i file *.c *.h che li riguardano.

2.1 KConsole

File: kconsole.c kconsole.h Funzioni: void kprint(const char*stringa,...);

Il funzionamento e' analogo a quello della normale printf tuttavia supporta solo i caratteri speciali

/n	A capo
/t	Tabulazione
%d	unsigned long rappresentazione decimale
%x	unsigned long rappresentazione esadecimale
%c	unsigned char carattere ascii (non stampa ascii con codice<32)
%s	char* stringa di caratteri

Attenzione! La lungheza massima della stringa _espansa_ non puo' superare il valore della macro KSTRNG_LUNG

2.2 Gestione Memoria

File: mem.c mem.h Funzioni:

```
mem_t mem_alloc(mem_t size);
bool mem_free(mem_t puntatore);
bool mem_resize(mem_t puntatore,mem_t newsize);

mem_t mem_totale();
mem_t mem_libera();
mem_t mem_usata();
bool mem_copy(mem_t porigine,mem_t pdestinazione,
                mem_t size);
bool mem_clear(mem_t puntatore,mem_t size);
```

La gestione della memoria si basa su due array: uno tiene conto delle aree di memoria libera, l'altro tiene conto di quelle occupate; entrambe hanno come Entry la struttura seguente:

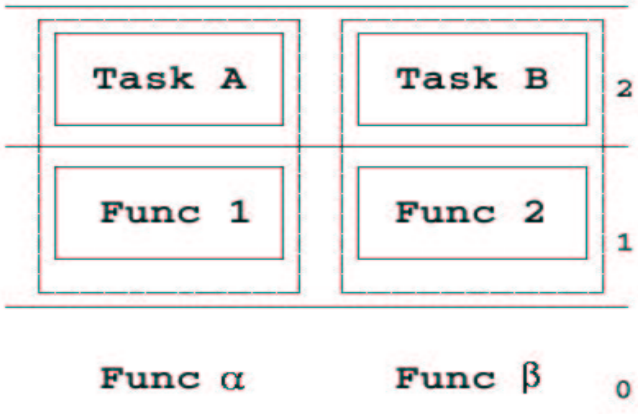
```
//un elemento della tabella di memoria
struct   mem_elemento_t
        {
                mem_t base;
                mem_t size;
                unsigned char flag;
        };
```

`base` e' l'inizio della zona di memoria, `size` e' la dimensione (entrabi in byte). `Flag` puo' assumere i valori 0 od 1 ed identifica se la entry e' valida oppure no.
I due array possono essere visti come due liste implementate attraveso array.
Quando avviene una richiesta di `<requestsize>` memoria, `mem_alloc()` scorre la lista di zone libera e trovatane una sufficentemente grande la riduce (o la cancella) e aggiunge una entry alla lista delle zone occupate.

Ogni tanto la memoria libera diviene talmente frammentata (rappresentata da tante piccole 'zone' adiacenti) che `mem_alloc()` non riesce ad allocare la memoria richiesta, sebbene questa sia libera. La funzione `mem_defrag()` supplisce a questo deframmentando la memoria libera, unendo cioe' molte piccole zone in una sola molto grande.

2.3 Livelli di Esecuzione

Per comprendere meglio la gestione dei Task di cheapOS parliamo prima dei suoi stati di esecuzione.



cheapOS in un qualunque momento si trova in uno dei suoi 3 possibile stati di esecuzione

(2) Modo Utente -- E' il modo con privilegi piu' bassi. L'area di memoria di codice, stack, dati e' limitata. Non si possono disattivare gli interrupt (attivi per default). A questo livello troviamo i task a livello utente.

(1) Kernel Mode (Task) -- Questo e' un modo di esecuzione intermedio. Ha quasi tutti i privilegi del modo Kernel 'Puro' ma e' comunque associato ad uno specifico Task, visto che ne utilizza lo stack. Inizialmente, a questo livello gli interrupt vengono disablitati, tuttavia e' possibile attivarli o meno utilizzando i comandi `cli` e `sti`. A questo livello troviamo Task che hanno chiamato una syscall oppure alcune routine di sistema (come ad esempio `idle`).

(0) Kernel Mode (Puro) -- E' il livello di massima esclusione e privilegio, usa uno stack indipendente e non e' associata a nessun Task. Anche se e' possibile, e' preferibile NON USARE MAI i comandi `cli` e `sti`! A questo livello troviamo routine di gestione degli ISR associati ad IRQ ed altre importanti routine interne di sistema.

Come vedremo cheapOS 'conosce' task gia' a livello di KernelCore, che, unita al sistema di gestione degli interrupt, predispone all'uso del Multitasking. Tuttavia la gestione dello scheduling e di altri aspetti piu' complessi sono demandati a strati di software superiori. Al KernelCore e' sufficente specificare i parametri del Task Corrente come vedremo in seguito.

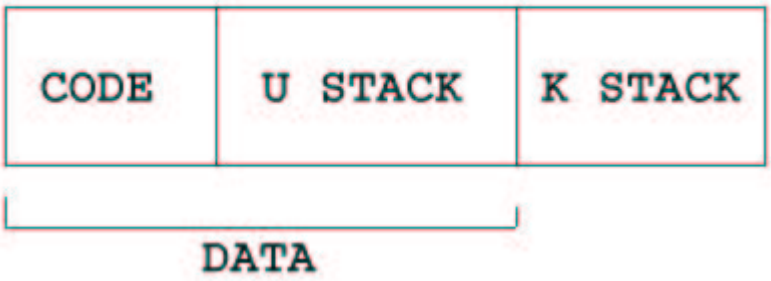
2.4 Gestione Task

File : task.h Funzioni:

```
bool task_init(struct task_t* task, struct task_mem_t* mem,
               int pid, unsigned int eip, int modo,
               unsigned int eflags);
```

task.c

```
void task_setcorrente(struct task_t* task);
void task_getcorrente(struct task_t* task);
void task_dumpscreen(const struct task_t* task);
void task_dumpscreen_current();
int task_dump(char* buffer,const struct task_t* task);
bool task_pusharg(struct task_t* task,char* arg);
```



Il KernelCore ha bisogno per funzionare correttamente che venga specificato ALMENO UN Task dimodoche' possa saltare agli stati di esecuzione 1 e 2.
Un Task e' rappresentato attraverso ad una struttura di tipo struct task_t:

```
//Struttura che identifica un Task
struct   task_t
         {
         signed int pid;
         struct task_mem_t mem; //Questi vengono fissati
                                //all'inizio e non cambiano!
         unsigned int modo;
         struct task_regs_t userRegs; //Questi cambiano durante
                                      //l'esecuzione
         struct task_regs_t kernelRegs;
         };
```

Una struttura task contiene inanzitutto 2 immagini del processore, ovvero due strutture **identiche** che rispecchiano i registri del processore, la prima `userRegs` ricorda lo stato del processore quando il Task e' stato interrotto mentre e' al livello 2, `kernelRegs` mentre e' in stato 1.
Naturalmente quando siamo al livello 0 non sono possibili interruzzioni, e comunque in quello stato non facciamo riferimento a nessun Task specifico!

```
//Stato del Processore
struct   task_regs_t
         {
         unsigned int eip,cs,eflags;
         unsigned int eax,ebx,ecx,edx;  //Presi dallo stato
                                        //del processore
         unsigned int esp,ebp,esi,edi;
         unsigned int ds,ss,es,fs,gs;
         };
```

La struttuta struct task_t contiene l'altrettanto importante riferimento ad un area di memoria partizionata come in figura. Particolarmente importanti sono i due segmenti di stack ovviamente uno per il livello 1 (`kstack`) l'altro per il 2 (`ustack`); il livello 0 e' provvisto di uno stack autonomo.
Nota che l'area dati corrisponde all'unione di quella codice e di quella `ustack`.
Attenzione: la zona di memoria a cui si fa riferimento non e' in alcun modo privilegiata e deve essere prima allocata con un `mem_alloc()`!

```
//Segmento di memoria del Task
```

```
struct   task_mem_t
         {
         mem_t base;
         mem_t codesize, ustacksize, kstacksize;
         };
```

Infine Task contiene anche le variabili modo che puo' assumere i valori KMODE od UMODE ad indicare lo stato corrente in cui si trova il task e l'intero (con segno) `pid` che contiene il pid del task.

La variabile `pid` non ha una particolare utilita' per il kernelcore, viene tuttavia messa a disposizione per poter identificare piu' facilmente un task da un altro in caso ce ne fossero molti.

KernelCore mette a disposizione alcune funzioni per manipolare piu' facilmente le struture task_t.

Facciamo un esempio:
Vogliamo allocare un task che abbia 20k di codice, 10k di stack utente ed altri 10k di stack kernel:

```
         //SONO AL LIVELLO 0 (per ora non importa di come ci sono
         //arrivato)

         struct task_t miotask;
         struct mem_t memoria_miotask;

         memoria_task.base=(mem_t)mem_alloc((20+10+10)*1024);

         //Mem Totale del Task = CodeSize+UStackSize+KStackSize

         if(memoria_task.base==0) goto Errore;
         //Non sono riuscito ad allocare la memoria che volevo!!

         memoria_miotask.codesize=20*1024;
         memoria_miotask.ustacksize=10*1024;
         memoria_miotask.kstacksize=10*1024;

         task_init(&miotask,&memoria_miotask,12,0,UMODE,0x202);
         //Inizzializzo la struttura miotask fornendo PID inizio del
         //codice eseguibile modo e Flags

         task_setcorrente(&miotask);
         //Setto il miotask come task corrente!

         //SALTO AL LIVELLO 2 (per ora non mi interessa come!)
```

adesso sto eseguendo miotask in modo 1 con flag 0x202 dalla istruzione 0 del codice!

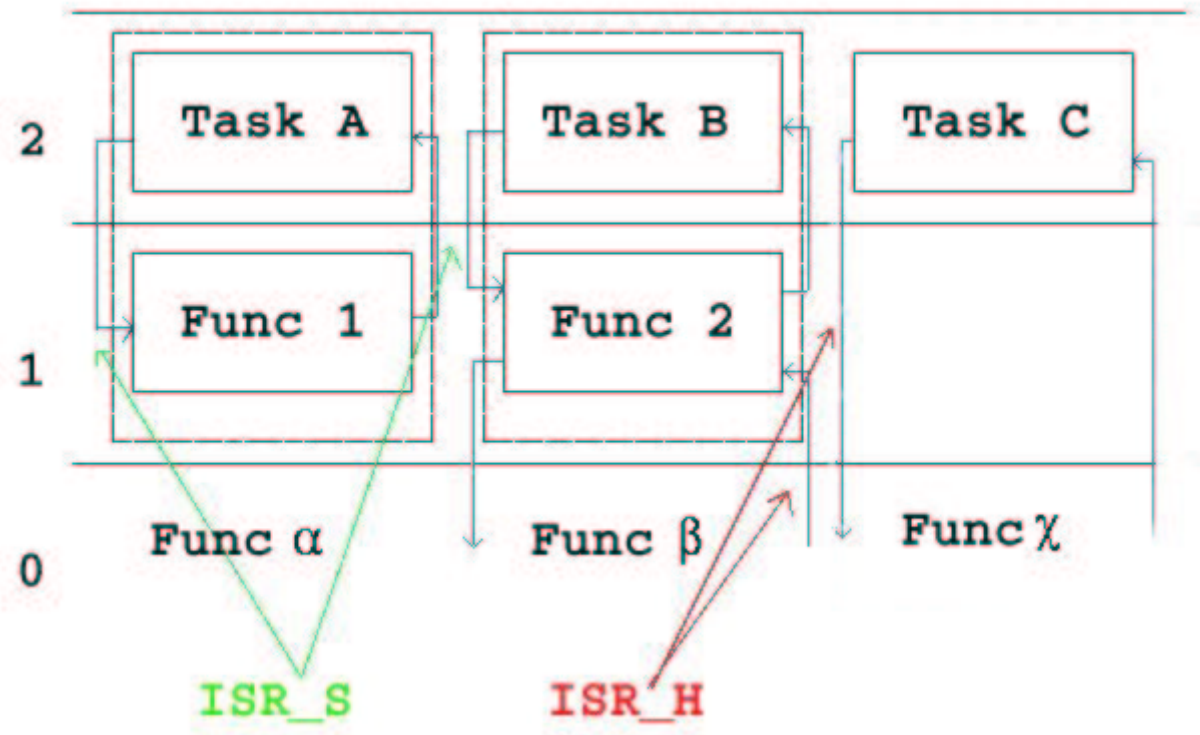
Potrebbe servire talvolta creare processi che non hanno mai bisogno di passare al livello 2 di esecuzione; casi tipo sono le routine del kernel.
E' quindi inutile allocare memoria necessaria per il modo 2!
`routine_kernel` e' semplicemente una funzione del kernel del tipo
`void routine_kernel(void);`

```
         //SONO AL LIVELLO 0 (per ora non importa di come
         //ci sono arrivato)

         memoria_task.base=(mem_t)mem_alloc(10*1024);
         //Mem Totale del Task = KStackSize
```

```
if(memoria_task.base==0) goto Errore;
//Non sono riuscito ad allocare la memoria che volevo!!

memoria_miotask.codesize=0;
memoria_miotask.ustacksize=0;
memoria_miotask.kstacksize=10*1024;

task_init(&miotask, &memoria_miotask, 12,
          (mem_t)routine_kernel,KMODE,0x202);
//Inizzializzo la struttura miotask fornendo PID inizio del
//codice eseguibile modo e Flags

//SALTO AL LIVELLO 1 (per ora non mi interessa come!)
```

adesso sto eseguendo il codice di kernel_routine() in modo 1 con flag 0x202!

2.5 Gestione IRQ-ISR e passaggio tra livelli di Esecuzione



File : irq.h Funzioni : void isr_setISR(unsigned char interrupt,void *handler);
isr.h
irq.c void irq_enable(unsigned short irq);
isr.c void irq_disable(unsigned short irq);
inline void irq_end();

Macro : ISR_DECLARE(nome_func)
ISR_CODE_H(nome_func)
ISR_CODE_S(nome_func)

ISR_ADDRESS(nome_func)

Ogni volta che accade un Interrupt un OS deve in quache modo gestirlo: cheapOS non e' da meno. In cheapOS gestire un INT corrisponde innanzitutto a saltare dal livello di esecuzione corrente ad uno piu' profondo (privilegiato), fare quel che si deve per rispondere all'interrupt, dopodiche' tornare ad un livello superiore (non necessariamente quello di partenza!).

Attenzione! Il livello 0 non ammette interruzioni visto che non e' possibile saltare ad uno piu' profondo! Se si dovesse presentare qui un interruzione la stabilita' del sistema risulterebbe compromessa, quindi e' necessario NON ABILITARE MAI AL LIVELLO 0 GLI INTERRUPT!!

Un pratico sistema di macro permette di scrivere routine per la gestione degli interrupt:
`ISR_DECLARE` dichiara che questa funzione gestira' un interrupt! Bisogna metterlo tra le dichiarazioni globali!
`ISR_CODE_H` questo codice appartiene ad una routine di livello 0
`ISR_CODE_S` questo invece ad una di livello 1
`ISR_ADDRESS` ritorna il puntatore alla ISR servira' per associarla con un particolare interrupt!

Facciamo un esempio: Voglio gestire la pressione di un tasto!

`miakeyboard.h`

```
#include "isr.h"
#include "irq.h"


void miakey_init();
```

`miakeyboard.c`

```
#include "miakeyboard.h"


ISR_DECLARE(miakey_handler);
//Questa func gestira' l'interrupt della tastiera

.....

//Inizializzio !
void miakey_init()
{

        .....
        isr_setISR(0x21,ISR_ADDRESS(miakey_handler));
        //0x21 e' l'interrupt associato alla tastiera
        //(linea IRQ 1)

        irq_enable(1);
        //Attivo la linea IRQ 1 della

        .....

        return;
}
```

```
        ISR_CODE_H(miakey_handler)  //Codice della mia ISR a livello 0
        {
                ....

                //Quello che deve accadere quando si preme un tasto!

                .....
                irq_end(); //Comunico alla PIC che puo'
                           //riattivare gli IRQ
        }
```

Quando adesso si preme un tasto, verra' eseguita la funzione `miakey_handler()` (se gli interrupt sono abilitati) e poi il sistema tornera' a fare quello che aveva interrotto.

Mentre le `ISR H` (livello 0) sono solitamente utilizzate per gestire Hw le `ISR S` (livello 1) servono per le syscall o interrupt software; in questo caso le cose sono un po' differenti.
In cheapOS il passaggio di parametri avviene attraverso i registri: se vogliamo passare dei parametri tra i livelli dobbiamo andare a leggere e scrivere i valori dei registri del task corrente. Questo puo' risultare complesso, pero' vedremo che vengono messi a disposizione del programmatore alcuni automatismi per implementare le syscall che semplificheranno notevolmente il lavoro.

Facciamo un esempio:
Voglio creare una syscall che, preso in ingresso un valore, lo elabori e lo ritorni al task utente.

miasyscall.h

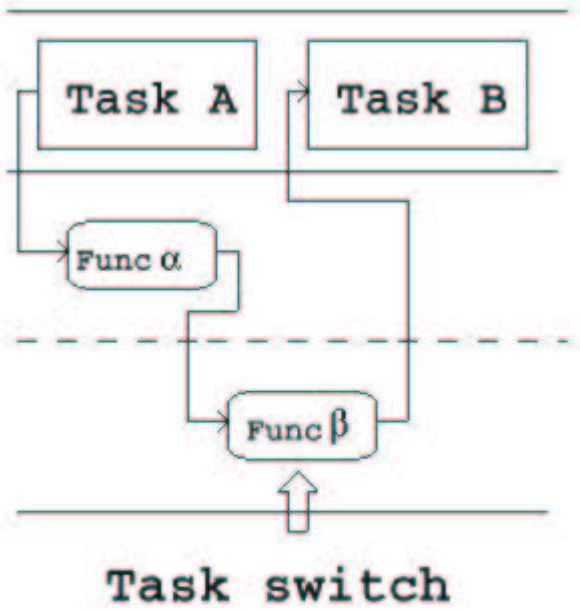
```
        #include "isr.h"
        #include "task.h"

        void miasyscall_init();
```

miasyscall.c

```
        #include "miasyscall.h"

        ISR_DECLARE(miasyscall_handler);    //Questa func gestira'
                                            //l'interrupt della tastiera
        .....
        //Inizializzio !
        void miasyscall_init()
        {
                .....
                isr_setISR(0x21,ISR_ADDRESS(miasyscall_handler));
                //0x45 e' un interrupt libero
                .....
                return;
        }

        ISR_CODE_S(miasyscall_handler)   //Codice della mia ISR
                                         //a livello 1
        {
                unsigned int ingresso,uscita;
                struct task_t task;

                task_getcorrente(&task);
```

```
            ingersso=task.userRegs.eax; //il valore del registro eax
                                        //e' il mio ingresso!
            ....

            //Quello che deve accadere quando un task
            //chiama questa syscall:
            //associo un valore alla var 'uscita'
            .....
            task.userRegs.ebx=uscita; //Il valore del registo ebx e'
                                      //la mia uscita!
            task_setcorrente(&task);
}
```

2.6 Task Switch



Il sistema di gestione degli interrupt attraverso 3 livelli introdotto in precedenza assieme a quello dei Task forniscono una base per l'implementazione del Multitask.

Il passaggio tra un task ed un altro consiste semplicemente nello scendere al livello 0, indipendente dal task attualmente in esecuzione, sostituire il task corrente e 'ritornare' ad un livello superiore che pero' adesso sara' riferito al nuovo task!

In pratica:

```
ISR_DECLARE(Tswitch);

struct listatask[2];
```

```
.......

ISR_CODE_H(Tswitch)
{
        struct task_t task;
        task_getcorrente(&task);
        //Copia in task lo stato del processo corrente

        //se e' il Task 0 salva il suo stato e passa al Task 1
        if(task.pid==0)
        {
                listatask[0]=task;
                task=listatsk[1];
        }

        //se e' il Task 1 salva il suo stato e passa al Task 0
        else if(task.pid==1)
        {
                listatask[1]=task;
                task=listatask[0];
        }

        //C'e' stato un Errore veramente brutto!
        else
                asm("hlt");

        task_setcorrente(&task);
}

......
```

Il codice sopra passa dal Task0 al Task1 o viceversa ogni volta che accade un interrupt , associando la ISR Tswitch ad esempio all'IRQ del clock. Avremo in questo modo una versione primitiva di Multitask. Vedremo tuttavia che cheapOS fornisce gia' un sistema di Multitask con molte funzioni.

Se dovesse essere necessario e' possibile effettuare un Task Switch anche al livello 1 di Esecuzione tuttavia in tal caso cheapOS non garantisce la coerenza dello stato 1 del processo che stava girando al momento dell'interrupt. SE POSSIBILE NON FARE MAI UNA COSA SIMILE!!

2.7 Gestione del Clock e TimeJob

File : time.h Funzioni: unsigned long long time_tick();
time.c void time_tm(struct time_tm_t* buffer);
unsigned long long time_tm2tick(struct time_tm_t* data);
void time_tick2tm(struct time_tm_t* data,unsigned long long tick);
unsigned char time_addTJ(unsigned long tickperiod,
unsigned int volte,time_job_t func);
bool time_isinvalid();

cheapOS fornisce alcune primitive per la gestione del tempo di sistema.
Fondamentalmente cheapOS conta il tempo in ticks, ovvero il numero di volte in cui la routine

principale di sistema viene richiamta; quindi visto che questo accade ogni 10 millisecondi possiamo affermare che 1 tick = 10 mlsec.

Durante lo startup il sistema acquisisce la data corrente dalla memoria CMOS la 'traduce' in ticks ed ad ogni interrupt del clock incrementa questo valore di 1.

Alcune funzioni permettono di conoscere il valore attuale di ticks e di tradurlo in tempo 'tradizionale' appoggiandosi alla struttura time_tm_t.
Ora come ora cheapOS non 'conosce' la data corrente ma solo l'ora del giorno!!

```
struct time_tm_t
{
        unsigned char ore;        //(0 - 23)
        unsigned char min;        //(0 - 59)
        unsigned char sec;        //(0 - 59)
};
```

Piu' importante della precedente e' la possibilita' di registrare funzioni che verranno poi eseguite ogni TOT numero di ticks per un numero prestabilito di volte; chiamamo queste funzioni TimeJob(TJ).

Ecco un esempio:

esempiotime.h

```
#include "time.h"

void init_esempiotime();
void TJ_uno();
```

esempiotime.c

```
#include "esempiotime.h"

.....
int conta;
.....

void init_esempiotime()
{
        .......
        conta=0;
        time_addTJ(6000,4,TJ_uno);  //ogni 6000 ticks quindio
                                    //circa ogni minuto!!!
        .......
        return;
}

void TJ_uno()
{
        // 'conta' e' variabile globale!!
        conta++;
}
```

La funzione TJ_uno() verra' richiamta per 4 volte ogni minuto; dopo 4 minuti la variabile 'conta'

avra' valore 4!
Se avessi voluto che la funzione venisse richiamata all'infinito avrei messo come secondo parametro 0 (ZERO).

ATTENZIONE: Attualmente i TJ vengono eseguiti al livello 0! Quindi non usare mai (direttamante o indirettamante) i comandi `cli` o `sti` in una di queste funzioni!

Capitolo 3

Scheduler

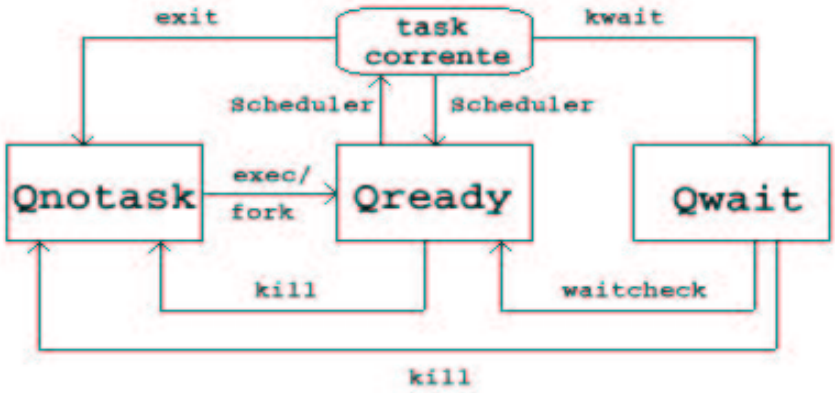
3.1 Code di PIDs Qnotask Qready Qwait

Scheduler di cheapOS

File : scheduler.h
scheduler.c

syscall_task.c

Funzioni : void scheduler();
syscall_task.h
inline void kwait(kwait_event_t func,unsigned long param);
int syscall_exit();
int syscall_fork();
int syscall_getpid();
unsigned long syscall_wait();
int syscall_waitpid(int pid);
int syscall_exec(char* nomefile,int argc,char* argv0);
bool syscall_kill(int pid);



cheapOS e' capace di Multitask: l'avvicendamento dei task e' basato su un array di strutture task_t che tiene conto dello stato dei Task in esecuzione e da tre code di PIDs (interi): Qnotask, Qready, Qwait.
La prima lista tiene conto dei PID liberi o indice delle strutture task_t inutilizzate, il secondo dei PID pronti o degli indici delle strutture che rappresentano Task pronti per essere eseguiti, l'ultima task in attesa di un evento.

NB: Il task di indice 0 e' riservato al processo di idle che viene richiamato ogni qualvolta la coda di ready risulta vuota!
Il PID del processo corrente e quello del task di idle non appaiono in nessuna coda!

La parte attiva dello scheduler consiste principalmente nella funzione/TimeJob scheduler() che richiamata ogni TOT ticks si comporta cosi':

- salva il processo corrente
- lo mette in coda di ready (a meno che non sia il processo di idle)
- estrae un nuovo processo dalla coda di ready (se e' vuota usa quello di idle)
- carica il nuovo processo come corrente

Alcune altre funzioni manipolano le medesime strutture e permettono di aggiungere o togliere task; la maggior parte di queste viene richiamta da getstore delle syscall che vedremo piu' avanti. Il loro funzionamento e' intuitivo. Semplicemente, mettono il processo corrente nella coda di Qnotaks per 'killarlo' o preso un pid da questa coda , allocano un task con il pid ottenuto e lo inseriscono nella coda di ready.

3.2 Processi in attesa coda Qwait e KWAIT_INT

In piu' lo scheduler ridefinisce anche una ISR H (di deafult associata al INT 0x42) per mandare i processi in coda di wait; funziona cosi:

- salva il processo corrente
- se e' in modo 2 ritorna (un processo puo' accedere a questa funzione solo se e' in modo 2)
- lo mette in coda di wait
- estrae un nuovo processo dalla coda di ready (se e' vuota usa quello di idle)
  carica il nuovo processo come corrente

Quando richiamiamo l'interrupt di wait e' necessario fornirgli una funzione ed un parametro che identificano un evento accaduto al quale il task messo in pausa puo' tornare in esecuzione.

La funzione e' del tipo `bool funzione_di_controllo(unsigned long parametro)`; finche' restituira' FALSE il processo rimarra' in coda di wait; appena restituira' TRUE, il processo' sara' accodato in Qready e non saranno fatti piu' controlli.

Il parametro e' un semplice intero (senza segno) che viene passato alla funzione di controllo: puo' rilevarsi molto utile!

Facciamo un esempio
Sono in modo1 e voglio che il mio processo rimanga in stato di wait finche' la variabile globale

'Controllo' non assumera' un valore voluto

```
unsigned int Controllo=1;

bool wesempio(unsigned int numero)
{
        if(Controllo==numero) return TRUE;
        return FALSE;
}

void miafunzione()
{
        .......
        ......
}
```

Quando verra' incontrata la funzione kwait il processo sara' messo in coda di wait ed altri tasks prenderanno il suo posto; il processo tornera' in coda di ready quando Controllo==12.
Attenzione! USARE `kwait()` _SOLO_ AL LIVELLO 1 !!

Cap 4 Gestione Filesystem a Volumi

File: filesystem.h
filesystem.c
syscall_filesystem.h
syscall_filesystem.c

Il solo modo che hanno i task di cheapOS per 'dialogare' con il mondo, cioe' avere accesso alle periferiche, e' il filesystem; anche l'i/o da tastiera/video avviene attraverso alcuni file 'viruali' cosi' come l'accesso ad informazioni riguardanti il kernel, tipo numero dei task in esecuzione data/ora corrente etc...

cheapOS supporta piu' filesystem contemporaneamente grazie ad una gestione unica di accesso ai file. Un file e' identificato da un PATH di ricerca contenente Volume, tipo di FS ed il file stesso.

Attualmente cheapOS supporta 3 filesystem

KER:	informazioni dal kernel
CON:	console (i/o a video)
FAT:	floppy (fat12 in lettura)

4.1 Volumi e PATH di ricerca (tabella delle sessioni)

Un PATH di ricerca di cheapOS e' cosi' fatto:
<VOLUME><PATH><FILE/DIR>

Il volume e' composto da 3 lettere (maiuscole) e da un ':' finale che identificano il tipo di filesystem e

la periferica (ovvero il volume).
Es 'FAT:'
Il path e' il percorso in cui si trova il file a cui vogliamo accedere

Es '\BOOT\GRUB'

Il file/dir e' il nome del file a cui vogliamo accedere

Es 'MENU.LST'

Es : 'FAT:\BOOT\GRUB\MENU.LST'

Attenzione: in cheapOS non vi e' distinzione tra file e directory se non per il fatto che le seconde terminano con il carattere di separazione '\'
Quindi : '\BOOT' e' il FILE BOOT
'\BOOT\' e' la DIRECTORY BOOT

Quando si apre una directory otteniamo un file (virtuale) al cui interno si trova la lista dei file contenuti in essa ed eventuali attributi (una stampa del comando ls, per intenderci). E' buona norma che tale file risulti **NON MODIFICABILE**! Se vogliamo modificare il contenuto della directory basta solo creare/cancellare i file/directory in essa contenuti.

Ogni volta che accediamo ad un file (indipendentemente dal fs) cheapOS ne tiene conto attraverso una tabella di strutture:

```
//Elemento che tiene conto del file aperto dal task (ovvero una sessione)
struct filesystem_accesso_t
{
        char nomefile[KSTRNG_LUNG];    //nome del file aperto
        int pid;                       //Task che accede al file
        int fs;                        //FileSystem Usato
        void* session;                 //sessione
};
```

Per ogni file 'aperto' viene tenuto conto del proprio PATH di ricerca, del pid del task che ha richiesto l'accesso del fs necessario (accelera gli accessi successsivi) e di un puntatore ad un tipo indefinito che puo' essere usato dalle funzioni del fs per 'passarsi' informazioni.

Attualmente cheapOS non fa' alcuna distinzione tra accesso in sola lettura ed accesso in scrittura; uno stesso file puo' essere aperto da un solo task alla volta, e per non piu' di una volta. I task figli di un padre che abbia accesso ad un file non ereditano alcunche'! Se un task termina prima di aver chiuso tutti i file aperti cheapOS pensera' a chiuderli automaticamente.

4.2 Come implementare un filesystem

cheapOS facilita l'aggiunta di nuovi volumi; ogni volume e l'implementazione di nuovi filesystem, consistono nel definire un nuovo elemento dell'array che contiene i volumi:

```
struct fsi_t
{
        char    nome[4];

        fsi_init_t init;
        //fsi_shutdown   shutdown;

        fsi_open_t      open;
        fsi_close_t     close;
        fsi_seek_t      seek;
        fsi_read_t      read;
        fsi_write_t     write;
        fsi_touch_t     touch;
        fsi_remv_t      remv;
};
```

Escludendo il primo elemento della struttura che identifica il nome del volume 'ESP:' i rimanenti sono indirizzi di funzioni del tipo specificato sotto.
Per <nomefile> viene passato il path del file troncato dell'identificativo del volume.
(Es. 'FAT:\BOOT\BELLO.TXT' --> \BOOT\BELLO.TXT').

```
typedef  bool    (*fsi_init_t)();
```

Viene invocata durante la sequenza di startup; serve ad inizzializzare il fs.
Attenzione e' a LIVELLO 0! Restituisce FALSE in caso di insuccesso!

```
//typedef bool(*f_shutdown)();
```

Attenzione: Attualmente la chimata Shutdown non e' implementata ne' usata! Dovrebbe venir invocata durante la sequenza di shutdown. Restituisce FALSE in caso di insuccesso!

```
typedef bool (*fsi_open_t)(void** session,char* nomefile);
```

Apre il file <nomefile> ed eventualmente alloca risorse 'appendendole' al puntatore ad indefinito `session` (passato per indirizzo); e' possibile usare i cast per allocare la struttura che necessaria. Restituisce FALSE in caso di insuccesso.

```
typedef bool (*fsi_close_t)(void** session);
```

Chiude il file rigurdante la sessione passata. Se possibile, liberare la memoria allocata per session (se non e' stato fatto) altrimenti andra' persa! Restituisce FALSE in caso di insuccesso!

```
typedef int(*fsi_seek_t)(void** session,int modo,int puntatore);
```

Scorre il file rigurdante la sessione passata secondo il modo <modo>:
MREL fa il seek di <puntatore> rispetto alla posizione corrente;
      <puntatore> puo' assumere valori negativi
MINIZIO fa il seek di <punatore> rispetto l'inizio del file (seek assoluto);
      <puntatore non ammette valori negativi>
MFINE fa il seek di <punatore> rispetto alla fine del file;
      <puntatore non ammette valori negativi>
in ogni caso viene restituito il puntatore alla posizione attuale assoluta (dopo il seek).

```
typedef int (*fsi_read_t)(void** session,int caratteri,
                          char* buffer);
```

Legge il file rigurdante la sessione passata per <caratteri> byte a partire dalla posizione corrente e li copia in buffer; il cursore viene incrementato dei bytes letti, in caso di errore restituisce invece il valore -1.

```
typedef  int(*fsi_write_t)(void** session,int caratteri,
                           char* buffer);
```

Scrive il file rigurdante la sessione passata per <caratteri> byte a partire dalla posizione corrente prendendoli da buffer; il cursore viene incrementato dei bytes scritti, in caso di errore restituisce invece il valore -1.

```
typedef  bool(*fsi_touch_t)(char* nomefile);
```

Crea il file/dir <nomefile>, restituisce FALSE in caso di insuccesso!

```
typedef bool(*fsi_remv_t)(char* nomefile);
```

Cancella il file/dir <nomefile>, restituisce FALSE in caso di insuccesso!

Per un esempio di filesystem fare riferimento al sorgente! Ci sono gia' tre esempi!

Cap 5 Gestione Syscall e Passaggio dei parametri

File: syscall.h
syscall.c

cheapOS definisce per syscall un unica ISR S (di default ad INT 0x41) sara' poi questa a richiamare le varie syscall 'particolari'.
Il passaggio di parametri avviene attraverso i registri del processore, in particolare si e' cercato di dare ad ogni registro un 'compito' particolare.

Registro	16 bit alti/16 bit bassi
eax	<Gruppo di syscall>/<Numero syscall>
ebx	<valore di ritorno> (in uscita)/<Parametro C> (in ingresso)
ecx	<Parametro A>
edx	<Parametro B>

Teoricamente con le syscall attuali cheapOS ha tutto cio' che gli serve, tuttavia se propio e' necessario non resta che aggiungere il 'case break' nella funzione 'syscall_handler()' nel file 'syscall.c'.

Nell' Appendice B si trovano i parametri associati ai registri per ogni syscall.

Capitolo 6

Driver dei Dispositivi

6.1 Introduzione ai dispositivi di input/output

cheapOS e' al momento in grado di ricevere input da tastiera e mostrare output sullo schermo in modalita' VGA a 16 colori, nonche' di gestire il drive floppy (in sola lettura con cache)

6.2 La tastiera

I driver e le funzioni associate alla tastiera sono contenute in quattro parti distinte del codice: keyboard.c/h si occupa dell'acquisizione vera e propria delle pressioni dei tasti, i quali poi vengono associati ad un layout di tastiera scelto dall'utente (disponibili adesso quello italiano e quello americano, rispettivamente in kbdlayout_ita.c e kbdlayout_usa.c), e infine keycoda.c/h che contiene un buffer). ed infine una parte che si occupa della gestione dei cosidetti 'combo', ovvero delle combinazioni di tasti alle quali si vogliono associare dei valori particolari. Passiamo a vedere in dettaglio cosa e' contenuto nelle varie porzioni di codice fino a qui menzionate.

keyboard.c/h e' il driver di tastiera vero e proprio.
Dopo aver definito un buffer di caratteri su cui si andranno ad eseguire le operazioni di scrittura (da parte della tastiera) e di lettura (da parte di applicazioni o altre parti dell'OS), viene dicharato un ISR per la gestione dell'input da dispositivo, gestione che e' poi demandata a ISR_CODE_H(kbd). Questa funzione si occupa di tradurre il valore prelevato dalla porta 0x60 (valore che rappresenta univocamente un tasto) in cio' che tale valore significa. Prima di proseguire bisogna precisare che la prima cosa che viene fatta durante l'inizializzazione della periferica e' il passaggio allo scan code modo 3 che invia per ogni tasto solamente un make e un break code, facilitando cosi' la gestione dei tasti premuti (gli altri metodi ne prevedono due o tre coppie per tasto).

La prima parte della funzione filtra i segnali 'particolari', ovvero tutti quei tasti che non sono caratteri alfanumerici; si possono individuare i seguenti segnali speciali:

- Reset della tastiera (se viene staccata e riattaccata).
- L'acknowledge a un comando inviato alla tastiera

- Il test diagnostico della tastiera (indica il corretto funzionamento o meno della tastiera)
- I tasti che comandano l'accensione/spegnimento dei led
- Il tasto ins (che richiama la funzione del video che imposta la dimensione del cursore video)
- I tasti modificatori (Ctrl, alt, shift ...)

Subito dopo questo filtraggio iniziale, si passa alla codifica del tasto (o della sequenza di tasti) premuti. Il codice verra' creato in questo modo: il byte meno significativo conterra' il carattere alfanumerico e il byte piu' significativo i vari modificatori (se presenti).
A questo punto avviene un secondo filtraggio: se il codice composto e' tale da rivelare la pressione di uno dei tasti tra Ctrl e alt, allora si assume che il codice possa corrispondere ad un 'combo' e si passa a scandire l'array delle funzioni combo registrate (vedi oltre). Nel caso il codice passi questo ulteriore filtro, cio' significa che e' stato premuto un carattere alfanumerico, il quale attraverso una scansione del layout attivo al momento, viene inserito nel buffer caratteri.

Le ulteriori funzioni presenti permettono di impostare o leggere i valori dei led, dello scan set e del layout di tastiera, o di estrarre un carattere dal buffer.

Durante la fase di inizializzazione vengono inoltre registrati cinque combo (vedi oltre) per passare tra le varie console e per abilitare/disabilitare la visione dei messaggi delle funzioni a schermo.

6.2.1 I layout

I layout di tastiera sono realizzati tramite array bidimensionali che associano un tasto (o dei 'combo' come Shift+tasto, Ctrl+tasto, ...) ad un codice ascii. La lunghezza della tabella e' attualmente di 256 caratteri, fatto che da' la possibilita' di usare alcuni dei codici che eccedono i 128 caratteri ascii standard per implementare funzionalita' quali lo scorrimento usando i tasti freccia, cancellare lo schermo o cambiare i colori... che avvengono qualora si legga dal buffer di tastiera uno dei valori ascii associati a queste funzioni.

6.2.2 I 'combo' (o combinazioni di tasti)

Da non confondere con le combinazioni che riguardano i caratteri ascii stampabili, i combo sono delle combinazioni di tasti che richiamano una particolare funzione. Possiamo registrare un combo per una funzione semplicemente usando la funzione `Kbd_add_shortcut()` a cui vengono passati come argomenti il valore codificato della combinazione di tasti e la funzione da richiamare. Alcuni esempi di tali combinazioni sono quelle che permettono di passare da una console all'altra premendo Ctrl+Alt+x (con x compreso tra 1 e 4) oppure la combinazione Ctrl-alt-canc per avviare lo shutdown del sistema.

6.3 Il video (VGA) e le consoles

Per gestire lo schermo, cheapOS lavora su due livelli: il primo e' il livello 'fisico' dove si pilota direttamente il contenuto dell'area di memoria riservata allo schermo (e gestita da graphics.c/h) mentre il secondo e' usato dalle consoles virtuali (console.c/h e fs_console.c/h).

- Il livello 'fisico'.
Gestito da graphics.c/h, a questo livello possiamo richiedere alla scheda video le informazioni sulla risoluzione dello schermo (che dovra' essere calcolata, in quanto non detta esplicitamente), impostare o leggere l'altezza del cursore video e la sua posizione all'interno dello schermo,

scegliere se volere o meno che gli sfondi possano lampeggiare (a discapito di una maggiore scelta di colori), scegliere il colore di primo piano e di sfondo di ogni posizione di carattere sullo schermo, nocnche' di pulire lo schermo.

- Il livello 'logico'.
Utilizzato da console.c/h (e da fs_console.c/h), questo livello si compone di 4 consoles virtuali dove e' possibile eseguire le normali operazioni che vengono effettuate a schermo, come la scrittura di caratteri e stringhe, lo spostamento del cursore, lo scroll e la pulizia dello schermo. In effetti queste quattro aree di memoria sono delle copie dell'area di memoria (anche la struttura e' la stessa, con ogni byte del carattere seguito dal byte degli atttributi (colori)) affinche' possano essere semplicemente copiate sulla memoria video per poterne mostrare il contenuto.
Questa operazione viene fatta da `Console_toScreen()` che si occupa appunto di copiare (usando `mem_copy()`) l'area riservata alla console virtuale sopra quella usata dalla memoria video.
Ogni console e' indipendente dalle altre e ogni task puo' scrivere su una o piu' consoles (sempre che siano disponibili e non utilizzate da altri processi!).
Il contenuto delle console e' visualizzabile a schermo tramite le combinazioni tasti Ctrl-alt-x dove x e' compreso tra 1 e 4.

Infine abbiamo la modalita' `MostraMessaggi`. Questa modalita' si e' rivelata molto utile nello sviluppo del sistema dopo l'introduzione delle console, che per loro natura (vedi `fs_console`) vengono ricopiate a schermo cancellando gli eventuali messaggi di informazione/errore derivanti da task o funzioni del sistema operativo. Questa funzione ridimensiona momentaneamente la console, rendendo la meta' inferiore dello schermo utilizzabile per visualizzare i messaggi di debug o di errore.
Questo avviene semplicemente limitando la console a meta' della sua dimensione disponibile, facendo cosi' in modo che quando viene scritta a schermo, la memoria video venga riscritta soltanto per meta', lasciando il contenuto della meta' inferiore intatto. Questa modalita' puo' essere disattivata in ogni momento per ripristinare l'interezza della console sullo schermo.

- Il filesystem.
Le quattro consoles disponibile vengono viste come files, e quindi gli si accede attraverso il filesystem.
Nei file fs_console.c/h si gestisce appunto l'accesso alle quattro console presenti. Durante l'inizializzazione vengono riservate quattro aree di memoria di grandezza pari a quella della memoria video (che verranno liberate durante la fase di shutdown del dispositivo) e a cui si avra' accesso tramite `Console_open()`.
Le azioni possibili sulle console sono quelle usuali di lettura e scrittura (gestite rispettivamente da `Console_read()` e `Console_write()`) e quella di scansione (`Console_seek()`).
Data la natura di file delle quattro console, l'accesso in scrittura da parte di piu' task e' gestito in maniera esclusiva (vedi Cap 4 - Gestione di filesystem a volumi).

6.4 Il floppy e la cache

File: floppy.h, floppy.c, cache.h, cache.c

Funzioni:
```
bool cache_init (struct cache_t* cache,int maxblocknum);
void cache_clear (struct cache_t* cache);

bool cache_findcopy    (struct cache_t* cache,int lba,
                        mem_t buffer);
```

bool cache_insertrefresh (struct cache_t* cache,int lba,
mem_t buffer);

Molto del codice per il driver del floppy e' ripreso dal SO Minirighi(TM) di Andrea Righi, tuttavia cheapOS implementa in piu' un sistema di cache in lettura.
La cache e' indipendente dal driver della periferica, purche' il dispositivo funzioni a blocchi di 512 bytes ognuno identificato da un 'lba' (linear byte address).

Ogni cache corrisponde ad un array, di grandezza arbitraria, composto dalla seguente struttura:

```
//Un Elemento della CACHE
struct cache_element_t
{
        int lba;
        int accessi;
        char    image[512];
}__attribute__ ((packed));
```

La cache funziona per frequenza: meno volte un blocco viene letto, piu' probabilita' ha di essere sovrascritto da un nuovo blocco!

La cache puo' essere facilmente aggiunta ad un qualsiasi driver cosi':

```
#include "cache.h"
........
struct cache_t miacache;
........
//Routine di Init del Driver
miodriver_init()
{
        ....
        cache_init(&miacache);
        ....
}
........

//Routine di Lettura del Driver
int miodriver_leggo(int lba,char*buffer......)
{
        .....
        if(!cache_findcopy(&maicache,lba,buffer))
        {
                //Ritorna con successo....
        }

        //Leggi Davvero!
        if(error)
        {
                //Ritorna con insuccesso!
        }

        //Inserisce o aggiorna il blocco nella cache
        cache_insertrefresh (&miacache,lba,buffer);
        .....
        //Ritorna con successo!
}
```

```
//Routine di Scrittura del Driver
int miodriver_scrivo(int lba,char*buffer......)
{
        //Scrive Davvero!
        if(error)
        {
                //Ritorna con insuccesso!
        }

        //Inserisce o aggiorna il blocco nella cache
        cache_insertrefresh       (&miacache,lba,buffer);
        .....
        //Ritorna con successo!
}
```

Appendice A

Tabella degli IRQ ed ISR predefinite

In cheapOS tutti gli IRQ sono definiti inizialmente e non possono essere modificati,alcune ISR sono predefinite e NON E' CONSIGLIABILE cambiarle!

Tabella IRQ

IRQ	Periferica o Funzione	Interrupt Associato	Usati da cheapOS(ISR definita)
00	Clock	0x20	X
01	Tastiera	0x21	X
02	?	0x22	
03	Seriale 2	0x23	
04	Seriale 1	0x24	
05	Hard Disk o Par. 2	0x25	
06	Floppy	0x26	X
07	Parallela 1	0x27	
08	Real Time Clock	0x28	
09	?	0x29	
10	?	0x2A	
11	?	0x2B	
12	? (forse mouse PS/2)	0x2C	
13	Coproc. Matematico	0x2D	
14	Hard Disk	0x2E	
15	?	0x2F	

Tabella ISR

Interrupt	Funzione	Tipo ISR
0x00	Divisione per Zero	H
0x01	Debug	H

0x02	Intel Reserved	H
0x03	Breakpoint	H
0x04	Overflow	H
0x05	BoundsCheck	H
0x06	Invalid Opcode	H
0x07	Coprocessor	H
0x08	Double Fault	H
0x09	Coprocessor	H
0x0A	Invalid TSS	H
0x0B	Segment	H
0x0C	Stack	H
0x0D	General Protection	H
0x0E	Page Fault	H
0x0F	Intel Reserved	H
0x10	Coprocessor	H
0x20	Clock	H
0x21	Tastiera	H
0x26	Floppy	H
0x41	Syscall	S
0x42	kwait(Riservato)	H

Appendice B

Tabella delle syscall

Syscall System (0x00)
Shutdown (0x00)

Registro	Ingresso	Uscita
eax	0x0000	?
ebx	?	bool
ecx	?	?
edx	?	?

Syscall Task (0x01)
Exit(0x00)

Registro	Ingresso	Uscita
eax	0x0100	?
ebx	?	?
ecx	?	?
edx	?	?

Fork(0x01)

Registro	Ingresso	Uscita
eax	0x0101	?
ebx	?	signed int
ecx	?	?
edx	?	?

GetPid(0x02)

Registro	Ingresso	Uscita
eax	0x0102	?
ebx	?	signed int
ecx	?	?
edx	?	?

Wait(0x03)

Registro	Ingresso	Uscita
eax	0x0103	?
ebx	?	unsigned int
ecx	unsigned int	?
Edx	?	?

WaitPid(0x04)

Registro	Ingresso	Uscita
eax	0x0104	?
ebx	?	signed int
ecx	signed int	?
edx	?	?

Exec(0x05)

Registro	Ingresso	Uscita
eax	0x0105	?
ebx	?	signed int
ecx	char	?
edx	char*	?

Kill(0x06)

Registro	Ingresso	Uscita

eax	0x0106	?
ebx	?	bool
ecx	signed int	?
edx	?	?

Syscall Filesystem (0x02)
Open (0x00)

Registro	Ingresso	Uscita
eax	0x0200	?
ebx	?	signed intl
ecx	char*	?
edx	?	?

Close (0x01)

Registro	Ingresso	Uscita
eax	0x0201	?
ebx	?	bool
ecx	signed int	?
edx	?	?

Seek (0x03)

Registro	Ingresso	Uscita
eax	0x0203	?
ebx	signed int	signed int
ecx	signed int	?
edx	signed int	?

Read (0x04)

Registro	Ingresso	Uscita
eax	0x0204	?
ebx	signed int	signed int
edx	char*	?
ecx	signed int	?

Write (0x05)

Registro	Ingresso	Uscita
eax	0x0205	?
ebx	signed int	signed int
ecx	signed int	?

edx	char*	?

Touch (0x06)

Registro	Ingresso	Uscita
eax	0x0206	?
ebx	?	bool
ecx	char*	?
edx	?	?

Shutdown (0x07)

Registro	Ingresso	Uscita
eax	0x0207	?
ebx	?	bool
ecx	char*	?
edx	?	?