Sistemi Embedded

Parte 2

RIVA 'ZAX' SAMUELE

Sommario – parte 2

- **Comunicazione**
  - Il caso dei microprocessori
  - Gerarchie di bus e protocolli
  - Bus IIC
- **Stima e gestione della potenza**
  - Introduzione
  - Come ottimizzare
  - ACPI
  - Stimare la potenza
- **DSP**
  - Introduzione
  - Architetture
- **Applicazioni Multimediali**
  - Introduzione
  - File Multimediali
  - Sistemi operativi
- **Linux – Laboratorio**
  - Kernel
  - Moduli
  - Proc FileSystem
  - Driver
  - Real Time

Comunicazione

Se il processore è fondamentale per *trasformare* i dati, la memoria per *salvarli*, la *comunicazione* viene implementata tramite bus. Un bus è un insieme di fili elettrici con una sola funzione, per questo si distingue in bus dati e bus indirizzi.
Processore, memoria e periferiche sono connesse al bus tramite delle porte, i *pin* dei diversi chip.

Per avere un efficiente trasporto dei dati si usano dei segnali di controllo attivati in base al tempo segnalato dal timer.

→ CASO DEI MICROPROCESSORI

Un microprocessore comunica con i device tramite alcuni dei suoi pin, secondo I/O basato sulle porte (come un normale registro) o sul bus (con porte di controllo, dati e indirizzi che permettono una comunicazione tramite determinati protocolli).
Se l'I/O è basato sui bus il processore dialoga con memorie e periferiche usando un singolo bus, usando però modalità differenti:

- memory mapped I/O: nello stesso spazio di indirizzamento vengono mappate una parte riservata alla periferiche e una parte per la memoria; non richiede istruzioni speciali;
- standard I/O: dei pin aggiuntivi definiscono l'accesso alla periferica o alla memoria [bus ISA].

Interrupts

Utili per evitare che un processore continui a controllare una periferica in attesa di dati (poll), quando è possibile attendere un segnale di interrupt che segnala la disponibilità dei dati.
L'implementazione degli interrupts richiede un pin extra "Int" che, se settato ad 1, invoca una ISR (Interrupt Service Routine).
Gli interrupt possono essere fissi (predefiniti nel processore) o vettoriali (specificati dalla periferica).
Un compromesso è la tabella degli indirizzi degli interrupt, che viene indirizzata dalle periferiche che specificano l'indice della riga in cui è contenuto l'indirizzo della ISR (che è modificabile).

Direct Memory Access (DMA)

Il microprocessore può cedere il controllo del bus di sistema al controller DMA in modo da poter eseguire processi senza doversi interrompere per gestire i dati temporanei salvati nei buffer, rendendo inutile l'utilizzo di certe ISR e quindi migliorando l'efficienza.

Arbitration

Come gestire più periferiche che richiedono un servizio dalla stessa risorsa?

- Inserisco processore single-purpose che fa da *priority arbiter* e definisce se una risorsa può o meno accedere alla risorsa.
  Le priorità possono essere fisse (ogni periferica ha priorità unica) o round-robin (variano in base allo stato precedente).
- La gestione può essere messa in mano alle periferiche aggiungendo della logica che le connetta tra loro formando una *daisy chain*, una catena in cui una periferica comunica con la risorsa, mentre le altre sono connesse tra loro. Eventuali richieste di risorse seguono la catena e di conseguenza avvengono gli assegnamenti, in base alla "vicinanza" della risorsa (equivalente della priorità).
  La catena è dinamica ma in caso di guasti può essere interrotta.
- Quando più microprocessori condividono un bus si usa una *arbitration orientata alla rete*.
  In questo caso si usa il protocollo del bus che gestisce gli accessi ed evita errori [usata on-chip].

→ GERARCHIE DI BUS E PROTOCOLLI

La presenza di tante periferiche che necessitano di una comunicazione veloce e frequente rallenta la qualità del servizio che sarebbe possibile ottenere con un singolo bus. Si introduce una gerarchia di diversi bus, uno dedicato al processore (alta velocità, comunicazione frequente), che connette i microprocessori/ cache/ controller della memoria, e un bus per le periferiche (più lento, di tipologia standard [ISA, PCI] per la portabilità), collegati tra loro da un bridge (processore single-purpose che converte la comunicazione tra i bus).
La presenza di più livelli di comunicazione permette di semplificare il protocollo e il trasporto di dati ai livelli più astratti (posso usare pacchetti invece che bit!).

A seconda della tipologia di livello fisico si possono avere diverse modalità di comunicazione:

- *comunicazione parallela*: è possibile trasportare più dati contemporaneamente (SCSI).
  Alto throughput su corte distanze, viene usato per connettere device sullo stesso IC (costoso);
- *comunicazione seriale*: si possono trasportare solo un bit alla volta ;
  Singolo cavo dati, adatto per lunghe distanze e poco costoso, comporta una maggiore complessità della logica e del protocollo (decompongo word in bit);
- *comunicazione wireless*: non esistono connessioni per il trasporto.
  Può avvenire tramire InfraRossi (corto raggio d'azione) o frequenze radio (onde elettro-magnetiche a distanza variabile).

Il rilevamento e correzione degli errori spesso fa parte dello stesso protocollo del bus, tipicamente la correzione avviene tramite ACK ed eventuale ritrasmissione.
Il rilevamento è possibile grazie ad una successione di bit di errore (ricevuti scorrettamente), usando bit di parità o checksum.

Protocolli seriali

- Molto usato è IIC (Inter-IC), un protocollo per bus seriali a 2 fili che permette la comunicazione tra circuiti integrati usando un semplice HW di comunicazione. Questa tipologia di protocolli è spesso usata da memorie Flash, clock real time, alcune RAM, microcontrollori. Max 3.4 Mb/s.
- Un'altro protocollo è CAN (Controller Area Network), usato per applicazioni real-time, per strumenti medici, fotocopiatrici... 11 bit di indirizzamento e max 1 Mb/s di velocità.
- Per alte performance si usa il protocollo firewire, sviluppato per interfacciare componenti elettronici separati e indipendenti (PC – scanner/ fotocamera), con plug&play.
  Il trasferimento dei dati arriva a 400 Mb/s grazie all'uso di pacchetti, indirizzamento a 64bit.
- Un protocollo che permette di fornire energia al device collegato è USB, trasmette a massimo 12 Mb/s, permette di collegare più device in una topologia a stella.

Protocolli paralleli

- Bus PCI (peripheral component interconnect), è un bus ad alte performance usato come standard industriale per connettere chip, board di espansione... ad alte velocità (fino 500Mb/s) con un'architettura sincrona e linee dati/indirizzo multiplexed.
- Un secondo protocollo è ARM bus, usato nei processori ARM, adatta la propria velocità al tempo di clock del processore (16 * X), con un indirizzamento a 64bit.

Protocolli wireless

- IrDA supporta trasmissioni ad infrarossi a corto raggio, trasferisce a max 4Mb/s, la mancanza di driver ne ha rallentato lo sviluppo nelle applicazioni.
- Bluetooth si basa su collegamenti radio a basso costo e corto raggio (10mt).
- IEEE 802.11 è lo standard per le LAN wireless, specifica i parametri per i livelli fisico e MAC della rete, funziona fino a 54Mb/s con la versione 802.11g.

→ *BUS IIC*

IIC nasce per fornire un controllo efficiente della comunicazione tra circuiti integrati. Per questo motivo tutti i dispositivi compatibili con questo bus hanno un'interfaccia on-chip.

Questo bus ha una linea dati (SDA) e una di clock (SCL), entrambe bidirezionali e connesse alla tensione di alimentazione tramite una resistenza di pull-up.
La comunicazione seriale tra i diversi dispositivi (hanno indirizzi decisi via software) è di tipo master-slave, usufruibile in tre modalità (standard, fast, high-speed a 3.4 Mb/s).

IIC è adatto per sistemi dominati dal controllo, non è necessaria una grande velocità e quindi si è scelto il collegamento seriale che consente di abbassare i costi anche se rende più complesso il protocollo. Il problema creato dai dispositivi con diverso clock viene risolto dando al master la possibilità di decidere il clock... in caso di più master esiste un meccanismo di arbitraggio sulla linea SDA (prima sui bit dell'indirizzo, poi sui bit dati, poi sull'ACK. Il master che perde un arbitraggio deve diventare subito slave per ricevere eventuale comunicazione).

Le condizioni START e STOP, effettuate dal master, sono effettuate tramite transizioni sulla linea SDA mentre SCL è HIGH, solo dopo un certo periodo seguente lo STOP il bus è considerato libero.
La rilevazione dei segnali avviene con semplicità se esiste un'interfaccia HW adeguata, in caso contrario viene usata un'interfaccia SW (può introdurre overhead).

Si possono inviare solo serie di 8 bit alla volta, a cui segue un bit di ACK da parte del ricevente (mette LOW la linea SDA, confermando la ricezione).
Nel caso in cui lo slave sia occupato a risolvere processi interni può mettere in attesa il bus, forzando a LOW la linea SCL dopo un ACK.

La sincronizzazione è possibile adattando il clock al device più lento (LOW minimo, HIGH massimo), tramite connessioni AND che attivano la ricezione solo quando tutti i dispositivi sono pronti.

Il trasferimento è generalmente composto da 7bit di indirizzo (inviati subito dopo lo START) + 1bit operazione + Xbit dati. Gli indirizzi sono composti da una parte fissa e una programmabile, esistono però 8 gruppi di bit riservati (0000*, 1111*: utili per broadcast, indirizzamenti alternativi o usi futuri).

Stima e Gestione della potenza

➔ INTRODUZIONE

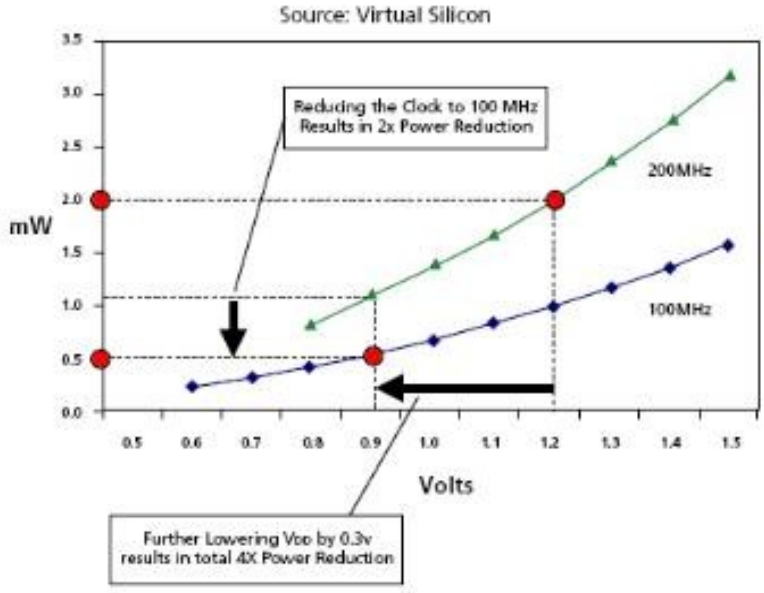
Il design a basso consumo di potenza riguarda il tempo di vita delle batterie, il raffreddamento e consumo energetico dei dispositivi ... è sempre più importante nello sviluppo dei sistemi VLSI (full-custom), per fronteggiare l'aumento esponenziale di transistor/CPU e delle frequenze.
Lo scaling del voltaggio dell'alimentatore permette di aumentare la densità dei transistor e la velocità delle operazioni. Molto importante è anche l'effetto dell temperatura, che al suo crescere causa consumi molto maggiori; ma ormai il più forte contributo alla dissipazione deriva dalla potenza statica di "leakage" (perdita) dovuta alla tecnologia sempre più miniaturizzata.

➔ COME OTTIMIZZARE

Esistono molte tecniche per massimizzare il risparmio energetico, a partire da un buon sviluppo di HW-SW codesign, uno scheduling adeguato, buoni algoritmi, un mappaggio intelligente dei componenti hardware, un adeguato raffreddamento dei circuiti...
I risultati migliori sono stati raggiunti dallo scaling della potenza dinamico, flussi di design e tool basati sul risparmio energetico, dai migliori architetture delle batterie... dal punto di vista HW si sono ridotte le correnti di switch e il voltaggio, così come le frequenze di clock. Tramite *Power Islands* si sono ridotte le perdite di circuito, dividendolo in isole "cluster" indipendenti (che si possono spegnere se non usati).

DPM e DVFS

Il PM dinamico cerca di gestire il consumo di potenza a run-time, implementando degli algoritmi nel sistema operativo che cercano di ridurre i consumi riducendo i consumi dei diversi componenti del sistema, adattando la frequenza e il voltaggio. Le modifiche di potenza portano ad un risparmio del 60%, ma usando DVFS (voltaggio dinamico e scaling delle frequenze) si adattano performance e consumi energetici in base all'uso dell'utente (risparmio 93%), via hardware e non via software.



L'abbassamento del voltaggio e della frequenza consente riduzioni quadratiche del consumo di potenza, perchè l'energia richiesta per eseguire un processo è | consumo_CMOS * Tempo| (il CMOS consuma proporzional-mente a V^2!!).

Nell'immagine a fianco si vede come abbassando la frequenza di clock e il voltaggio si ottenga una riduzione dei consumi del 200% (4 volte).

L'adattamento dinamico di questi fattori avviene via hardware (reagisce alzando la frequenza in risposta ad attività della CPU) o via HW-SW (dopo un periodo di learning si adatta la frequenza ai consumi).

Ulteriori miglioramenti possono derivare da studi più approfonditi: dato che una sequenza di esecuzione di un programma consiste in istruzioni CPU e istruzioni di memoria, si deve tenere conto del fatto che la modifica di frequenza non incide (quasi) nelle istruzioni di memoria (stallo in attesa dei dati). In certi processori si inserisce una PMU (performance monitoring unit) che genera statistiche e adatta la DVFS in base al tipo di istruzioni.

L'implementazione hardware di DVFS consiste nel inserimento di un circuito di controllo del voltaggio on-board, oltre ad un DAQ (data acquisition) che misura la potenza istantanea.
In certi casi l'abbassamento della frequenza comporta una diminuzione della durata delle batterie, perchè i sottosistemi del computer (memorie, periferiche...) aumentano la propria durata dei tempi di esecuzione, a causa della minor frequenza di clock (minor aggiornamento). E' necessario quindi trovare un compromesso tra frequenza e tempi di aggiornamento in modo da raggiungere ottime prestazioni, sempre tenendo da conto anche un'eventuale perdita di performance.

→ OTTIMIZZAZIONE DEI SISTEMI DIGITALI

L'ottimizzazione del consumo di potenza a livello sistema consiste di diverse tecniche attuate in diversi livelli di astrazione e in diversi momenti della progettazione.

- Partizionamento HW-SW: tramite un flusso di progetto adeguato che tenga conto sia del lato SW che di quello HW si possono ottimizzare le performance tenendo conto di vincoli di potenza.
  In base alle specifiche sono possibili approcci a task-level (modello funzionale, creo grafo con processi e dipendenze) o behavioral-level (modello strutturale, ho componenti HW, la specifica è in linguaggio HL). In base a metriche di tipo costo/performance viene stabilito il partizionamento.
- Ottimizzazione HW: dipende da:
  * scelta della rappresentazione dei dati (complemento a 2 ad esempio);
  * "codifica su bus": diminuzione dell'attività di switch sul bus (consuma molta potenza), possibile sia sui dati che sugli indirizzi tramite algoritmi come gray_code [non ridondante], T0_code [ridondante, aggiunge un bus INC per max velocità e PM], Working_zone [indica indirizzi come id+offset che si riferiscono ad un registro addizionale, più bus], Beach_code [usato se i precedenti sono inappropriati, trova correlazioni spaziali/temporali tra indirizzi], Info_theoretic_code [implementazione complessa, semplifica il problema della codifica basandosi sulle correlazioni]. Complicano i circuiti.
  * "design della memoria": memorie e cache dissipano molta energia, ottimizzando la gerarchia di memoria e l'accesso si può minimizzare il consumo. Si cerca di rafforzare la località dei dati tramite replicazione dati [implicita o esplicita tramite buffer tipo victim-cache] e partizionamento [cache multi-bank indipendenti, possono essere messe in sleep].
  Altre ottimizzazioni puntano a rendere più denso il codice per minimizzare l'occupazione di memoria, tramite InstructionSet alternativi [servono compilatori adeguati], e compressione del codice oggetto [full o selective, che indicano se comprimere tutte o parte delle istruzioni... in modo da ottimizzare la tabella IDT usata per decomporre il codice].
  * "codifica delle istruzioni": si punta a minimizzare il (basso) consumo di potenza di un processore che esegue un certo tipo di applicazioni, agendo sulla circuiteria di fetch & decode.

→ ACPI

Dal 1996,con ACPI, si è iniziato a parlare di PM anche lato software. ACPI è usato sui sistemi desktop, nei sistemi embedded invece generalmente si usa una gestione ad hoc oppure si lascia la scelta al programmatore.
ACPI è l'evoluzione di APM che si basava sul BIOS, di cui migliora la robustezza e le funzionalità, implementando nel sistema operativo le politiche di gestione della potenza.

ACPI specifica solo l'interfaccia tra HW e SW, è composto da ACPI Tables (fornite dal firmware, contengono pseudocodice AML che permettono all'OS di manipolare le funzioni di PM dell'hardware), Bios e Registers.
ACPI consente di mettere il sistema in 4 stati differenti (working, sleeping, soft off, mechanical off), i primi due con sottostati che differiscono per consumo di potenza. A seconda del livello si possono variare certi parametri come la luminosità dello schermo, la portata del bluetooth, clock della CPU... inoltre gestisce le batterie e la temperatura ((dis)attiva ventola).

Per l'hardware ACPI-compatibile sono previsti 2 modelli: fixed (indica la posizione dei registri a cui l'OS può accedere) e generic (OS effettua chiamate AML per agire sui registri).
ACPI riceve le informazioni sulle capacità di PM dalle System Description Tables, tramite cui viene costruito l'ACPI Namespace, una struttura gerarchica contenente riferimenti agli oggetti dell'architettura ACPI.

Da ACPI sono nate diverse tecnologie atte al miglioramento del PM: Intel Speedstep (controllo dei consumi nei portatili), OnNow (gestione PM)...
Nei sistemi embedded la gestione della potenza è fondamentale e dev'essere considerata in ogni passo della realizzazione del sistema. In realtà pochi RTOS hanno un sottosistema di PM, tra questi ci sono OS-9 e Windows CE (ha 4 stati possibili, run, partial-power, zero-power, standby).

→ STIMA DI POTENZA

Un passo cruciale per l'analisi di un sistema embedded è lo studio del consumo di potenza tramite meccanismi di stima, sia relativi all'hardware che al lato software.

Lato HW
Un modello hardware base (PowerPlay) per calcolare la potenza dissipata da un componente tiene conto della capacità media e della corrente passata dalla fonte di energia. In realtà servono modelli più avanzati e dinamici che tengano conto del carico di lavoro, un primo miglioramento si basa su Power FSM.
PFSM è modello guidato dagli eventi il cui sistema è composto da un gestore di memoria, delle risorse e una batteria. Il modello consiste in più stati (con consumo energetico X) e più transizioni (con tempi "performance" Y), il gestore della potenza riceve le richieste e gestisce i passaggi tra gli stati in modo avere buone performance a consumi minimi.
Lato SW
Lato software si usa design space exploration per simulare e confrontare le diverse specifiche, i microprocessori, il rapporto performance/ latenza/ consumi e scegliere quindi le caratteristiche più adatte al sistema in progettazione.
I consumi di potenza lato SW sono dovuti al sistema di memorie (cache miss, località dei dati), ai bus (accessi decisi run time lato SW), al data-path (via SW sono decise le istruzioni da eseguire).

Esistono diversi livelli di astrazione: livello Gate (usato per benchmark, è accurato ma lento), livello RT (a partire dalle stime dei consumi dei blocchi di alto livello di un processore [ALU, memorie] calcolo i consumi totali medi), livello istruzione (misuro direttamente la corrente in uscita dalla fonte di potenza mentre eseguo certe istruzioni, simulo le istruzioni in HDL, ognuna con un proprio costo. Ho una parte dinamica basata sullo stato precedente che aumenta flessibilità del modello tenendo conto degli effetti dovuti ad una serie di istruzioni).

Uso un approccio funzionale a livello istruzione per raccogliere delle statistiche avanzate di dati sia a livello intra-processore che inter-processore.
Questo modello è adatto per un singolo processore (modello lineare che permette di avere un ridotto insieme di misure di istruzioni -> learning-set, per poi estrapolare l'energia di tutte le altre istruzioni) ma anche per più processori (faccio media tra i parametri stimati, aumenta però la varianza al crescere dei processori).
Questo approccio seleziona 5 funzionalità (fetch,branch,scrivi registriALU,load+store) e analizza i diversi consumi di potenza. Successivamente si è migliorato il processo per architetture superscalari o con pipeline, aggiungendo parametri come il livello di parallelismo e l'overhead per gli stalli.

Progetto POET
Il progetto POET divide l'analisi in due livelli: livello sorgente (creo albero di istruzioni atomiche

caratterizzate da un tempo di exec e da un consumo di potenza medio X), livello assembler (basato su compilatori e modello architetturale particolare). Allo stesso tempo si introducono calcoli data-dependent, ottenuti tramite analisi delle librerie e del sistema operativo, usando modelli statistici.

DSP

➔ INTRODUZIONE

Un Digital Signal Processor è un tipo di microprocessore decisamente veloce e potente, usato in molti prodotti industriali, militari, medici...

L'architettura dei DSP è composta di diversi algoritmi e dispositivi che lo rendono così performante:

- moltiplicatori veloci: algoritmi le cui operazioni ottimizzate sono add & mult, si aggiunge un componente HW dedicato alla multiply-accumulate (MAC);
- più unità di esecuzione: operano in parallelo e possono eseguire processi complessi rapidamente;
- efficiente accesso alla memoria: per eseguire operazioni MAC serve accesso ai dati in un solo ciclo, quindi si usano rapide memorie di programma (collegate direttamente alla cache istruzioni) e dati con proprio bus. Dato che l'accesso a memoria di questi processori è facilmente predicibile si usano unità che le generano in anticipo;
- cicli a zero overhead: non sono usati contatori per i cicli e non ci sono branch;
- instruction set specializzato: viene usato il più possibile l'hardware, minimizzando lo spazio di memoria necessario per memorizzare programma DSP. Per questo motivo i DSP non sono programmati in linguaggi tipo C, si usa assembler che permette maggiori ottimizzazioni.

➔ ARCHITETTURE DSP

- Negli anni '80 esistevano architetture con una solo unità MAC, un'ALU, poche unità di esecuzione. Successivamente si è aggiunta I-Cache, pipeline profonde e PM più avanzato.
- I processori DSP più avanzati odierni invece eseguono più MACS per ciclo di clock, hanno Instruction Set avanzati, ampi bus dati e più unità di esecuzione.
- Esistono architetture multi-issue che usano istruzioni semplici e permettono l'issue ed esecuzione parallela di gruppi di istruzioni (VLIW da 4-8 istruzioni a ciclo, decise dal programmatore).
Sono costosi e consumano molta energia, non sono adatti a certe applicazioni.
- Trami tecniche SIMD (single instruction, multiple data) si aumentano le performance, dando al processore la possibilità di eseguire in parallelo istruzioni uguali su dati differenti.

I DSP sono i processori a miglior compromesso di performance, consumo di potenza e prezzo, inoltre sono a disposizione molti tool di sviluppo. Solo in certe applicazioni si usa un ibrido DSP/ microcontrollore, più adatto alle applicazioni che necessitano di controllo.

Applicazioni Multimediali

→ Introduzione

Le applicazioni multimediali trattano dati relativi ad almeno due media continui (streaming video, videogiochi...) riprodotti in un certo intervallo di tempo.
Queste applicazioni sono caratterizzate da un data rate elevato, riproduzione real-time (descritte con parametri relativi alla QoS).

→ File multimediali

Il filesystem dei sistemi multimediali deve tener conto di formati di file molto differenti tra loro (audio, video, testo...) che sono acquisiti in modo diverso ma devono restare sincronizzati tra loro.
- Audio: l'onda sonora viene trasformata dal microfono in segnale elettrico analogico, poi campionato e convertito in digitale (PCM-cd audio). La compressione usa diversi algoritmi tra cui mp3.
- Video: si sfrutta la persistenza dell'immagine sulla retina per codificare appropriatamente il video (max 50 immagini/sec), usando standard (PAL, NTSC) che usano interlacing per ridurre il numero di frame al secondo (25-30). Se il video è codificato in digitale si usa RAM Video per raggiungere anche gli 80 frame/sec. La compressione è necessaria per ridurre la banda, si usa JPEG o MPEG (sfrutta ridondanze spaziali e temporali).

→ Sistemi operativi

I SO multimediali differiscono dagli altri per 3 componenti fondamentali:

- scheduling dei processi: si usa uno scheduling real-time (RMS, eDFS) oppure (raramente) uno scheduling tra processi omogenei (RR a tempo).
  RateMonotonic è statico, assegna priorità fissa ai processi in base alla frequenza di esecuzione.
  eDeadline First è dinamico, si basa sulla dichiarazione di ogni processo della propria scadenza.
  In genere in un VideoServer si usa RMS (più semplice) se ci si trova entro il limite per cui si è certi di rispettare i vincoli, altrimenti eDFS.
- filesystem: il sistema per cui è necessario usare syscall per interagire con il FS non è adatto, i requisiti RT sono più stretti, quindi si usa un Video Server che invia in continuo i blocchi dati che saranno gestiti dal client. Per il caso di video on demand ogni utente passa tra stream video differenti che sono condivisi e ottimizzati in base alle richieste.
  Il partizionamento del disco è fondamentale per migliorare la QoS, i file sono molto grandi e ci si accede in sequenza, si mettono porzioni di file video,audio e testo in uno stesso blocco per evitare seek multipli sul disco fisso. Uso modello SmallBlock (uso RAM durante la riproduzione) o LargeBlock (più frame per blocco, ci può essere spreco di spazio per la frammentazione).
  Anche il posizionamento di più video su un VideoServer è importante, tramite legge di Zipf si classificano in base alla popolarità per fare in modo che non ci siano troppi spostamenti delle testine (algoritmo a canne d'organo che mette al centro del disco i film + popolari).
  Per maggiori prestazioni si usano dischi paralleli e politiche di caching alternative.
- scheduling del disco: l'esigenza di elevati data rate e la possibilità di richieste multiple da più utenti impone l'adozione di algoritmi particolari per lo scheduling del disco:
  * statici: lo scheduler si basa sul concetto di round che sfrutta la predicibilità di stream simili. Si usa double buffering per mantenere un elevato data rate.
  * dinamici: usato per stream differenti poco predicibili, viene specificata una deadline e quindi usato un algoritmo detto scan-EDF, che serve le k richieste a deadline più vicina con ordine in base al numero di cilindro.

Linux

➔ KERNEL

Introduzione
Il kernel controlla e media l'accesso all'hardware, implementando astrazioni di basso livello quali i file, i processi, i device... inoltre schedula e alloca le risorse del sistema, rafforza la sicurezza e la protezione verso dati critici.
L'obiettivo di un kernel è ottenere alte performance, robustezza, flessibilità e sicurezza lasciando allo stesso tempo all'utente una libertà di costruire ed implementare nuove funzionalità di più alto livello, tramite le interfacce "system call", le chiamate di sistema.

Linux Kernel
Linux è un sistema operativo open source basato su UNIX, con multitasking preemptive, memoria virtuale, librerie condivise, moduli del kernel dinamici, protocolli di rete TCP/IP e supporto al SMP.
Il kernel di Linux è solo uno dei 4 sottosistemi principali del sistema operativo (applicazioni-servizi-kernel-controller HW) e si può considerare come una macchina virtuale che media l'accesso hardware delle diverse applicazioni, gestendo l'accesso alla CPU in modo equo basandosi sull'algoritmo di scheduling scelto.

Il kernel Linux è composto da 5 sottosistemi:
- Lo scheduler dei processi (SCHED): è diviso in 4 moduli (policy di scheduling; moduli specifici e non dell'architettura in uso che comunicano con la CPU o la MM per sospendere o riattivare i processi; l'interfaccia di sistema che vincola l'accesso alle risorse).
- Il gestore della memoria (MM): è composta di 3 moduli (moduli specifici e non dell'architettura gestiscono accesso alla memoria fisica e a quella virtuale, gestendo ad esempio i page faults; interfaccia di sistema che fornisce un accesso vincolato ai processi).
- Il filesystem virtuale (VFS): presenta una vista consistente dei dati salvati sui componenti hardware, fornendone un'astrazione e supportando diversi formati di FS. Diversi moduli (un driver e un FS logico per ogni FS supportato, 3 interfacce di accesso).
- L'interfaccia di rete (NET): supporta molti protocolli e componenti hardware, in modo tale che i processi possano accedervi in modo trasparente (un driver per ogni device, 2 interfacce, moduli per ogni protocollo di rete supportato).
- La comunicazione tra processi (IPC).

Tutti i sottosistemi dipendono dallo SCHED, perchè necessitano di sospendere e attivare processi, MM viene usata per supportare meccanismi di condivisione di memoria e per il context switch.
Ogni sottosistema contiene delle informazioni di stato a cui è possibile accedere tramite determinate interfacce.

Le strutture dati sono Task List (blocchi di dati divisi per processo, salvati in una lista concatenata a cui lo scheduler accede per indicare il blocco attualmente in uso), I-Nodes (nodi indice che rappresentano logicamente i file in un filesystem, memorizzano le posizioni fisiche in cui si trovano i dati reali).

Codice sorgente e Linux 2.6
Il codice sorgente del kernel contiene l'implementazione dei 5 sottosistemi, ma una grande porzione di spazio è riservata ai drivers. Il kernel 2.6 semplifica la compilazione ed introduce grandi novità tra cui l'hyperthreading (un singolo processore, mascherandosi a livello hardware, si mostra ai procesi come 2 o più processori), la preemption dei task che girano in modalità kernel, lo scheduler O(1) per migliori prestazioni e load balancing.

Il boot del sistema
Dopo l'accensione viene attivata via hardware una zona di memoria predefinita che esegue il BootLoader (LILO/Grub) che carica il "primo programma", cioè il kernel e precisamente il processo */sbin/init*.

Le chiamate di sistema
Sono le interfacce tra i processi e l'hardware, permettono un certo grado di portabilità dei programmi aggiungendo funzionalità di sistema e sistemi di astrazione.
Le API POSIX forniscono esplicite richieste verso il kernel tramite interrupt (chiama *int $0x80* che porta la CPU in modalità kernel). Un gestore di syscall salva i registri e lo stato del sistema, esegue quindi la chiamata a seconda di una tabella di dispatch statica presente nel kernel.

→ MODULI

Introduzione
I moduli sono "pezzi" di codice che vengono aggiunti al kernel a run-time, sono parte del kernel e comunicano direttamente con il *base kernel* (quella parte del kernel contenuto nell'immagine avviata).
I moduli Linux (LKM) sono modulari (se no che moduli sarebbero? :-D) perchè il supporto precompilato ad ogni elemento HW è sconveniente, si preferisce un modello più flessibile in cui è possibile aggiungere moduli senza ricompilare ogni volta l'intero kernel.

I LKM sono usati per driver di device/network/fs, chiamate di sistema... essendo parte del kernel ne condividono il code space e possono minarne la sicurezza e stabilità, anche perchè funzionano con i privilegi di root.

Nella pratica...
I LKM sono file di tipo ELF, se non sono inclusi nel codice del kernel hanno delle proprie procedure e vengono inseriti nel sistema tramite il comando *insmod* o *modprobe* (se voglio verificare le dipendenze), oppure rimossi con *rmmod*.
Un modulo viene "linkato" appena caricato, non possono usare funzioni di libreria ma solo chiamate di sistema.

Visualizzo i moduli tramite *lsmod* e vedo informazioni con *modinfo*. Nel proc filesystem in */proc/modules* si trovano informazioni sui moduli caricati.
In */etc/modprobe.conf* si trova la configurazione.

Ogni modulo contiene la routine di inizializzazione *init_module()*, chiamata automaticamente dopo *insmod*, e la funzione *clean_module()*, eseguita dopo la rimozione.
Per passare parametri, dal kernel 2.6 è possibile specificarne il numero, si usa una forma di questo tipo: *module_param (nome, tipo, num, perm)*.

Nel caso venga caricato nel kernel un modulo che genera degli errori, viene generato un messaggio oops che viene interpretato da certi programmi come ksymoops e aiuta nel debug (si usa kdb).
Un altro tool interessante è UML, una virtual machine su cui far girare software buggato in modo che sia possibile usare gdb senza rischiare di installare kernel non stabili e insicuri.

Nel kernel 2.6 si è aumentata la sicurezza e stabilità del sistema (opzione per unload dei moduli, framework per device drivers, supporto per plug&play e PM...).

→ PROCFS

Proc è un filesystem virtuale che esiste solo in memoria, i cui file permettono a programmi utente di accedere a determinate informazioni del kernel, oltre ad avere utilità di debugging (basta usare il file header *linux/proc_fs.h*).
Tramite certe routine è possibile creare strutture dati o link simbolici nel procFS (*create_proc_entry()* o *proc_symlink()*), creare directory (*proc_mkdir()*), leggere/scrivere dati (*read_func()* o *write_func()*).

→ DRIVERS

Introduzione

Un driver è un'interfaccia tra il kernel e l'hardware che permette la comunicazione e lo scambio di dati, unifica l'accesso ai device dello stesso tipo, proteggendo l'accesso all'hardware da potenziali usi errati. Ogni componente fisico viene così rappresentato da file a cui si può accedere tramite syscall standard (read,write...).
Da un altro punto di vista i driver sono uno strato software tra le applicazioni e i device fisici, tramite driver si decide come mostrare l'hardware al livello superiore e come gestire le diverse situazioni.
Essendo il primo strato software del computer, sono necessarie protezioni per rendere i driver stabili e sicuri, si deve semplificare il debug e aumentare la flessibilità senza però complicare troppo lo sviluppo e il codice.

I driver possono funzionare in diverse modalità: tramite polling o interrupt (alcune funzioni devono però essere completate senza interruzioni, in quel caso disabilito gli INT). Se supportato, si può usare DMA per trasferire i dati dalla memoria senza occupare cicli di CPU.

...in Linux

I device sono elencati nella cartella */dev* (porte seriali, usb, HD, terminali, interfacce di rete...). Una lista dei device usati la si trova in */proc/devices*, dove vediamo i componenti a carattere (a stream, ci si accede sequenzialmente: console, porta seriale), a blocchi (accesso "random", dati trasferiti in blocchi, permette il mount del sistema e la bufferizzazione), interfacce di rete (comunicazione via pacchetti).

Driver e relativi device sono identificati da numeri *major* (1-254, usati dal kernel per eseguire istruzioni sul driver corretto) e *minor* (0-255, permette di identificare il singolo device), di 8 bit ciascuno. Ad esempio, si registra un driver con *insmod nome.o*, si crea un device con *mknod /dev/chardev c 254 0* (major = 254, minor = 0);

Scrivere driver

All'avvio un driver cerca il device a cui fa riferimento e quando lo trova si registra, tramite la prima routine chiamata dopo *insmod,* cioè *init_module()*, che esegue le operazioni preliminari e associa il driver ad un numero major (tramite *rmmod* si invoca cleanup_module() che rilascia il major).
Le operazioni (fops) che un driver può eseguire su un device sono indicate in <*linux/fs.h*>, un array di puntatori alle funzioni (metodi) che agiscono sui file (oggetti). Tra le filo operation indichiamo *llseek()* (cambia la posizione di lettura/scrittura in un file), *read()*, *write*() (leggo/scrivo dall'address space del kernel tramite speciali funzioni in <asm/uaccess.h>, in caso di page fault il processo entra in sleep; si usano semafori per controllare l'accesso alle zone critiche di memoria), *readdir()* (legge directory), *poll()* (controlla se un device è leggibile o scrivibile), *ioctl()* (permette di eseguire comandi specifici del device), *mmap()* (richiede il map della memoria di un componente nell'address space), *open()* (inizializza il device se appena aperto, alloca i dati), *release()* (opposto di open(), dealloca una struttura dati, "chiude" eventualmente il device) e *lock()* (lock sul file).

→ LINUX REAL TIME

Linux non è un sistema Real-Time, non ha una gestione rapida degli interrupt e del context switching, è senza garanzie nello scheduling dei processi.
E' necessario cambiare le regole di scheduling (no time-sharing, uso FIFO o FIFO-RR, con priorità e preemption), cosa possibile tramite la syscall *sched_setscheduler()*.
Sempre in user space è necessario dare la possibilità di fare lock su certe zone di memoria tramite *mlock()* o *mlockall()*.

Per aumentare la predicibilità della latenza bisogna invece spostarsi nel kernel space, dove viene introdotta la gestione diretta degli interrupt hardware. Tramite kernel timer si aumenta la granularità del sistema e permette l'esecuzione di processi periodici.