



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

*Appunti di
Sistemi Operativi II*

Anno 2021

Valentino Bocchetti

Contents

1 Introduzione al corso	18
1.1 Esame	18
1.1.1 Appelli giugno/luglio	18
1.1.2 Appello di settembre	18
1.1.3 Appelli successi	19
2 Lezione del 08-03	19
2.1 I sistemi operativi	19
2.2 Sistemi mono e multi programmati	19
2.3 Sistemi paralleli	20
2.4 Sistemi distribuiti	20
2.5 Ambiente di elaborazione (Client-Server o Peer-to-Peer)	20
2.6 Cloud computing	21
2.7 Sistemi mobile	21
2.8 Avvio di un sistema operativo nel dettaglio	21
2.9 Tabella degli interrupt	22
2.10 Memoria centrale	22
2.11 Memoria cache	22
2.12 Memoria di massa	22
2.13 I processi	23
2.14 Activity	24
2.15 Politiche utilizzate dallo scheduler	25
2.16 Sistema mono e multi tasking	25

2.17 Throughput (utilizzo della CPU)	26
2.18 Overhead	26
3 Lezione del 12-03	26
3.1 Sincronizzazione dei processi	26
3.2 Comunicazione con i dispositivi	28
3.3 File system	28
3.4 La shell dei comandi	29
3.5 La memoria virtuale	29
3.6 Memoria virtuale in Linux	30
3.7 Spazio di indirizzi di un processo	31
4 Lezione del 15-03	33
4.1 Swapping	33
4.2 Memory Arbiter	33
4.3 La RAM nei sistemi UNIX	34
4.3.1 Risultati desiderati	34
4.3.2 Stack & heap	34
4.4 Memory Management Unit (MMU)	36
4.5 Segmentazione	36
4.6 Indirizzo logico	36
4.6.1 Segment selector	36
4.6.2 Offset	37
4.6.3 Descrittore di segmento	37
4.6.4 Traduzione degli indirizzi logici in indirizzi fisici	38

4.6.5	Privilegi di esecuzione	39
4.6.6	Tabelle dei Descrittori di Segmento	39
4.7	Segmentazione in sistemi LINUX	40
4.7.1	I segmenti in Linux	41
4.7.2	Spazio degli indirizzi lineari	41
4.7.3	Pagina di indirizzi lineari	41
4.7.4	Segmentazione vs Paginazione	41
4.7.5	Page	41
4.7.6	Tipi di Paginazione	42
4.7.7	TLB	43
5	Lezione del 19-03	43
5.1	La programmazione di sistema	43
5.1.1	System call	43
5.1.2	Fork	44
5.1.3	Processo Zombie	44
5.1.4	Esempio	44
5.2	Gestione dei processi in Linux	44
5.3	Process Descriptor	45
5.4	Stati da un processo	45
5.5	Identità di un processo	45
5.6	Relazioni fra processi	45
5.7	Le liste di processi RUNNING	45
5.8	I file hash	46

5.8.1	Funzione di accesso h	46
5.8.2	Divisione	46
5.8.3	Risoluzione delle Collisioni	46
5.8.4	Tabella Hash per la runqueue	46
5.9	Organizzazione dei processi	47
5.9.1	Wait queue	47
5.9.2	Struttura di una Wait queue	47
5.10	Duplicazione delle risorse	47
5.10.1	Chiamate di sistema in Linux	48
5.11	Bomba fork	48
5.12	Terminazione di un processo	48
5.13	I thread	49
5.13.1	Kernel-level e user-level thread	49
5.13.2	Clone()	50
5.13.3	Pthreads	50
6	Lezione del 22-03	50
6.1	Kernel thread	50
6.1.1	Processi	51
6.1.2	Process switch	51
6.2	Scheduling dei processi	52
6.2.1	Scheduling policy	52
6.2.2	Single e multi-core	52
6.2.3	Tipologie di Scheduling	53

6.2.4	Caratteristiche di uno scheduler	53
6.2.5	Prelazione	54
6.2.6	Algoritmo di scheduling	54
6.2.7	Scheduling di processi convenzionali	55
6.2.8	Quanto temporale di base	55
6.2.9	Durata del quanto temporale di base	55
6.2.10	Average sleep time	56
7	Lezione del 26-03	56
7.1	Processi batch e interattivi	56
7.2	Scheduling dei processi Real-time	57
7.3	Strutture dati usati dallo scheduler	57
7.3.1	Principali campi della struttura	57
7.4	Scheduling dei processi nel dettaglio	58
7.5	Funzioni utilizzate dallo scheduler	59
7.6	Aggiornamento del time slice di processi Real-time	59
7.7	Rendere un processo running	60
7.7.1	Funzione schedule()	60
7.7.2	schedule() prima e dopo il context switch	61
7.8	Gestione dei Page Frame	62
7.9	Page Descriptor	62
7.9.1	Campi di un Page Descriptor	62
7.10	NUMA	62
7.11	Zone di memoria	63

7.12 Zoned Page Frame Allocator	64
7.12.1 Per-CPU page frame cache	65
7.13 Kernel Mapping di page frame	65
7.14 Il problema della frammentazione esterna	65
7.15 Il Buddy System	65
7.15.1 Filosofia del Buddy System	66
7.15.2 Strutture dati	67
7.15.3 Liberazione di un blocco	67
8 Lezione del 29-03	67
8.1 Per-CPU Page Frame Cache	67
8.2 La funzione <code>__alloc_pages()</code>	68
8.3 La funzione <code>__free_pages()</code>	68
8.4 Memory Area Management	69
8.5 Lo Slab Allocator nel dettaglio	69
8.6 Cache Generiche e Specifiche	70
8.7 Allocazione e Deallocazione di Slab	72
8.8 Gli Object Descriptor	72
8.9 Slab Coloring	72
8.10 Local Cache e Memory Pool	73
8.11 Processi utente e gestione della memoria	73
8.12 Lo spazio degli indirizzi	73
8.13 Regioni di memoria	74
8.14 Strutture dati	74

8.15 Ricerca di una regione di memoria	75
8.15.1 Alberi red-black	75
8.16 Linux e la ricerca di una regione di memoria	75
8.17 Diritti di accesso ad una regione di memoria	75
8.18 Demand Paging	76
8.19 Page Reclaiming	76
8.20 Selezione di una pagina target	76
8.20.1 Principi alla base di un PFRA	77
8.21 Periodic Reclaiming	78
9 Lezione del 31-03	78
9.1 Virtual File System (VFS)	78
9.1.1 Common File Model	79
9.1.2 Strutture dati del VFS	79
9.1.3 L'oggetto superblock	79
9.1.4 Operazioni sul superblock	80
9.1.5 L'oggetto inode	80
9.1.6 Il File Object	80
9.1.7 Dentry File	81
9.1.8 Tipi di filesystem	81
9.1.9 La gestione dei filesystem	81
9.1.10 I namespace	81
9.1.11 Filesystem Mounting	82
10 Lezione del 09-04	82

10.1 Event Driven	82
10.2 Time Driven	83
10.2.1 Problema 1 → Osservabilità degli eventi	83
10.2.2 Problema 2 → Ritardo di rilevazione	83
10.2.3 Problema 3 → Ordine di occorrenza	84
10.3 WatchDog	84
10.4 Task periodici e Aperiodici	84
10.5 Deadline	84
10.6 Scheduling Real-Time	84
10.7 Rate Monothonic Scheduler	84
10.7.1 Test di schedulabilità	85
10.8 EDF Scheduler	85
10.9 Sistemi Soft e Hard Real-Time	86
10.9.1 Gestione di task in contesti ibridi	86
10.9.2 Limiti dei comuni sistemi Real-Time	86
10.9.3 Desiderata per i Sistemi Real-Time	86
10.9.4 Prevedibilità	87
10.10 I linguaggi di programmazione	88
10.10.1 Real-Time Euclid	88
10.10.2 Real-Time Concurrent C	88
10.11 Sistemi Embedded	89
10.11.1 Caso Semplice I	89
10.11.2 Caso Semplice II	89
10.11.3 Caso Complesso I	89

10.11.4 Caso Complesso II	89
10.11.5 Arduino	90
10.11.6 Microcontroller	90
10.11.7 La scheda Arduino	90
10.11.8 Programmare in Arduini	90
10.12 RT Linux	90
10.13 RTAI	91
11 Lezione del 16-04	91
11.1 Virtualizzazione	91
11.2 Macchina virtuale - Definizione	91
11.3 Definizione della virtualizzazione	91
11.4 Vantaggi	92
11.5 Tipi di Virtualizzazione	92
11.5.1 Emulation	92
11.5.2 Full Virtualization	93
11.5.3 VMM di sistema - realizzazione	94
11.5.4 Paravirtualization	95
11.5.5 Operating System level Virtualization	95
11.6 Architettura VMware	96
11.7 Virtualizzazione delle risorse	96
11.8 Cloud computing	96
11.8.1 I precursori - Grid Computing	97
11.8.2 I precursori - EDGeS	97

11.8.3 Service-Oriented Architecture (SOA)	97
12 Lezione del 19-04	98
12.1 Tipologie di Grid	98
12.2 Dal Grid al Cloud: l'evoluzione	98
12.3 Cloud Vs Grid: la rivoluzione	98
12.4 Cloud Computing	98
12.5 Infrastrutture IT onsite (senza cloud)	99
12.6 Cloud IT infrastructure	99
12.7 Thin client	99
12.8 Riferimenti	99
12.9 Risorse	100
12.10 On Premise	100
12.11 Scaling	100
12.12 Filoni Tecnologici nel Cloud computing	100
12.13 Conoscenze informatiche che contribuiscono al Cloud Computing	101
12.14 Cloud computing (definizione)	101
12.15 Software as a Service (SaaS)	101
12.16 Platform as a Service (PaaS)	102
12.17 Infrastructure as a Service (IaaS)	102
13 Lezione del 26-04	102
13.1 Dispositivi palmari	102
13.2 Smartphone	103
13.2.1 I sistemi operativi	103

13.2.2 La memoria	103
13.2.3 L'alimentazione	103
13.2.4 Periferiche	103
13.2.5 Garanzie dei SO mobili	103
13.3 Pattern Architetturali	104
13.3.1 Modello MVC (Model View e Controller)	104
13.4 Symbian	104
13.4.1 Struttura del kernel	104
13.4.2 Hardware	104
13.4.3 MMU	105
13.4.4 Gestione della memoria	105
13.4.5 I thread	105
13.4.6 Lo scheduling	106
13.5 Iphone e IOS	106
13.5.1 Struttura IOS	106
14 Lezione del 30-04	107
14.1 Struttura di un sistema IOS	107
14.2 Mach	107
14.2.1 Messaggi	107
14.2.2 Invio e ricezione di mesaggi	107
14.2.3 Le porte	108
14.2.4 Privilegi delle porte	108
14.2.5 Primitive di sincronizzazione	109

14.2.6 Mutex	109
14.2.7 Read/Write block	110
14.2.8 Semafori	110
14.2.9 L'oggetto host	110
14.2.10 L'oggetto clock	110
14.2.11 Processor object	110
14.3 IOS - scheduling	111
14.4 Scheduling primitives	111
14.5 IOS - I thread	111
14.6 IOS - I task	112
14.7 IOS - le API di Task e Thread	112
14.8 IOS - Le priorità	112
14.9 IOS - Le run-queue	113
14.10 IOS - Le wait-queue	113
14.11 IOS - Lo scheduler	114
14.11.1 Handoff	114
14.11.2 Continuation	114
14.12 IOS - la prelazione	114
14.13 Le policy	114
14.14 Internet of Things (IOT)	115
14.14.1 Di cosa di tratta	115
14.14.2 Diffusione	116
14.14.3 Ambiti Applicativi	116
14.14.4 Sensori e Smart Objects	116

14.14.5 Discipline e attività di ricerca collegate alla IoT	116
15 Lezione del 03-05	117
15.1 Android OS	117
15.2 OHA	117
15.3 Nascita di Android	118
15.3.1 Le versioni di Android	118
15.4 Struttura del sistema	120
15.5 Applicazioni	120
15.5.1 Application framework	121
15.5.2 Le librerie	121
15.5.3 Runtime	121
15.5.4 Il Kernel	121
15.5.5 Applicazioni	122
15.5.6 Gli intent	122
15.5.7 Le Activity	123
15.5.8 I Services	124
15.5.9 Content Provider	125
15.5.10 Broadcast Receiver	125
15.6 Processi e Thread	126
15.6.1 Multitasking	126
15.6.2 Stack delle activity	126
15.7 Lo scheduling dei processi	127
15.7.1 CFS	127

15.8 La gestione della memoria	127
15.9 Il power management	127
15.10 Il wakelock	128
16 Lezione del 07-05	129
16.1 Problemi legati al timekeeping	129
16.2 Wakeup events framework	129
16.3 Wake source object	130
16.4 La sicurezza in Android	131
16.4.1 Vulnerabilità - Accesso Fisico	131
16.4.2 System Partition and Safe Mode	131
16.4.3 Sicurezza a livello kernel	132
16.4.4 Modello di sicurezza in Android	132
16.4.5 Protezione a livello di applicazioni	132
16.4.6 Le Sandbox	133
16.4.7 Protezione a livello di applicazioni	134
16.4.8 Lo User ID	134
16.4.9 Lo Shared User ID	134
16.5 I permessi	135
16.6 Meccanismi di controllo	136
16.7 La crittografia	136
16.8 Protezione del filesystem	137
17 Lezione del 10-05	137
17.1 Le biometrie	137

17.2 Qualificazione delle Biometrie	138
17.3 Comparazione di tecnologie biometriche	138
17.4 Caratteristiche Biometriche	139
17.5 Sicurezza dei Sistemi biometrici	139
17.6 FingerPrint	139
17.6.1 Vantaggi e Svantaggi	140
17.6.2 Metodo di Matsumoto	140
17.7 Retina	140
17.7.1 Vantaggi e Svantaggi	141
17.8 Geometria della mano	141
17.9 Firma	141
17.9.1 Vantaggi e Svantaggi	142
17.10 Riconoscimento vocale	142
17.10.1 Vantaggi e Svantaggi	143
17.11 Riconoscimento dell'orecchio	143
17.11.1 Vantaggi e Svantaggi	143
17.12 Riconoscimento del volto	144
17.12.1 Vantaggi e Svantaggi	144
17.13 Iride	144
17.13.1 Polarizzazione dell'immagine	144
18 Lezione del 14-05	145
18.1 Iride - Matching	145
18.2 Iride - nel dettaglio	145

18.3 Metodi di rilevazione dell'iride	146
18.4 NICE - Noisy Iris Challeng Evaluation	146
18.5 MICHE I & MICHE II	146
18.6 Biometrie soft	146
18.6.1 Tobii 1750	146
18.7 Ambiti di ricerca	147
18.8 Fasi di un tipico sistema biometrico	147
18.9 Struttura di un riconoscitore biometrico	148
18.9.1 Fase di Enrollment	148
18.9.2 Fase di Testing	148
18.10 Approcci alla detection	148
18.11 Approcci alla recognition	149

1 Introduzione al corso

Ricevimento (da concordare almeno 24 ore prima) → Venerdì 10-12 (chat privata su teams)

Email → silviobarra@unina.it con oggetto [INFO-S02] (iscritto al *anno di corso e corso di laurea* SO2 esame facoltativo o obbligatorio)

Obiettivi

- Scelte progettuali ed implementative di un reale sistema operativo a base linux
- Sistemi distribuiti e del cloud computing
- Sistemi operativi ad uso mobile
- IoT

Prove intercorso:

- Domande sul sistema linux
- Domande sui sistemi mobile

1.1 Esame

Prima parte fino ai SO Real-Time (compreso) + Seconda Parte (il resto)

1.1.1 Appelli giugno/luglio

Lo studente si inscrive all'appello intero, riceve due votazioni (uno per parte)

1. Se i singoli voti sono entrambi maggiori o uguali a 18, lo studente potrà registrare l'esame;
 - Il voto finale sarà la media dei 2 voti ottenuti (arrotondato per difetto);
 - Lo studente potrà chiedere un orale per migliorare (o peggiorare) il voto;
2. Se entrambi i voti sono minori 18, lo studente dovrà effettuare entrambe le prove in uno degli appelli successivi, con le stesse modalità
3. Se uno dei 2 voti non supera il 18, o lo studente rifiuta una delle 2 parti, potrà sostenere solo la relativa prova di uno degli appelli successivi di **giugno/luglio/settembre**

1.1.2 Appello di settembre

Lo studente che dovrà fare l'esame completo, riceverà un voto unico (non si conservano più le singole parti)

Lo studente che dovrà fare una delle due prove farà solo quella che gli manca

- Nel caso dovesse fallire la prova o rifiutare il voto ottenuto, verrà invalidata anche la parte precedentemente ottenuta;
- Se dovesse ottenere un voto maggiore o uguale a 18 sarà a descrizione del docente la richiesta di una prova orale

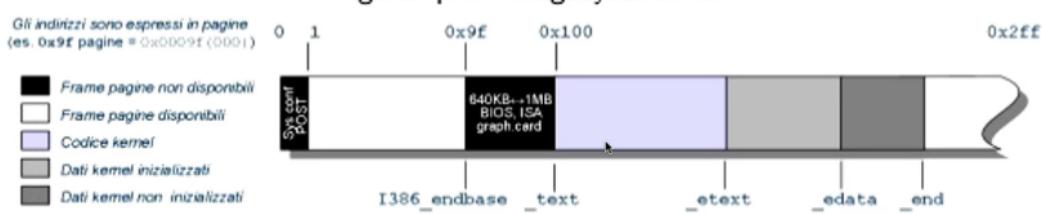
1.1.3 Appelli successi

Lo studente farà l'esame completo e riceverà un voto unico (non si conservano più le parti)

2 Lezione del 08-03

2.1 I sistemi operativi

Le componenti di un sistema di calcolo è così definita:



Un **sistema operativo** è difficile da definire (in base alle grandi variazioni che presenta). Una definizione è quella di definirlo come un insieme di software atti a gestire l'hardware sottostante.

Può essere visto come un alzacalciatore di risorse:

- memoria;
- dispositivi di I/O;
- processi

Può essere visto come un programma di controllo:

- gestore dei programmi di sistema ed applicativi;
- verifica del corretto funzionamento dei dispositivi di I/O

2.2 Sistemi mono e multi programmati

Un sistema è detto **mono-programmato** quando in ogni momento è presente una sola **immagine** (pacchetto processo + dati)

Con l'introduzione del concetto di **spooling**, nascono i primi sistemi **multi-programmati** (il sistema è in grado di eseguire più job in concorrenza). Ciò comporta di tener conto della gestione delle risorse ad essi dedicate:

- Tempo di calcolo
- Memoria
- I/O

In questi sistemi si ha il caso di sistemi multi-utenti. Di conseguenza nascono i sistemi **time-sharing** (nascita dei primi PC). Ogni processo utilizza periodicamente un intervallo di tempo prestabilito (un **quanto**)

2.3 Sistemi paralleli

Si arriva alla nascita dei sistemi **Paralleli** dotati di più CPU :

- Symmetric multiprocessing (SMP) → ogni processore esegue una copia identica del SO
- Asymmetric multiprocessing (AMP) → a ciascun processore è assegnato un task specifico; il processore **master** schedula e allora il lavoro per i processori **slave**

2.4 Sistemi distribuiti

Ripartiscono l'elaborazione su diversi processori. Ciascun processore ha una memoria locale e i processori comunicano con gli altri attraverso metodi di comunicazione quali bus ad alta velocità o rete. La rete può essere:

- Locale (LAN);
- Geografica (WAN);
- Metropolitana (MAN);
- Small (SAN).

2.5 Ambiente di elaborazione (Client-Server o Peer-to-Peer)

Esistono due principali tipi di server:

- server per l'elaborazione;
- file server

Nei sistemi Peer-to-Peer invece ciascun pc può essere server o client

- Il carico è suddiviso su più server (si evitano i sovraccarichi);
- Necessita una politica di gestione
- Nuova entità (**load balancer**)

2.6 Cloud computing

Paradigma di uso delle risorse computazionali attraverso il cloud (memorizzazione, archiviazione ed elaborazione dati)

2.7 Sistemi mobile

Il sistema operativo in questo caso ha orientamento specifico (a causa delle dimensioni del dispositivo)

2.8 Avvio di un sistema operativo nel dettaglio

All'avvio (o al reset) di un sistema il programma di **bootstrap** presente sul firmware inizializza i componenti del sistema, carica ed avvia il SO, che rimane in attesa di:

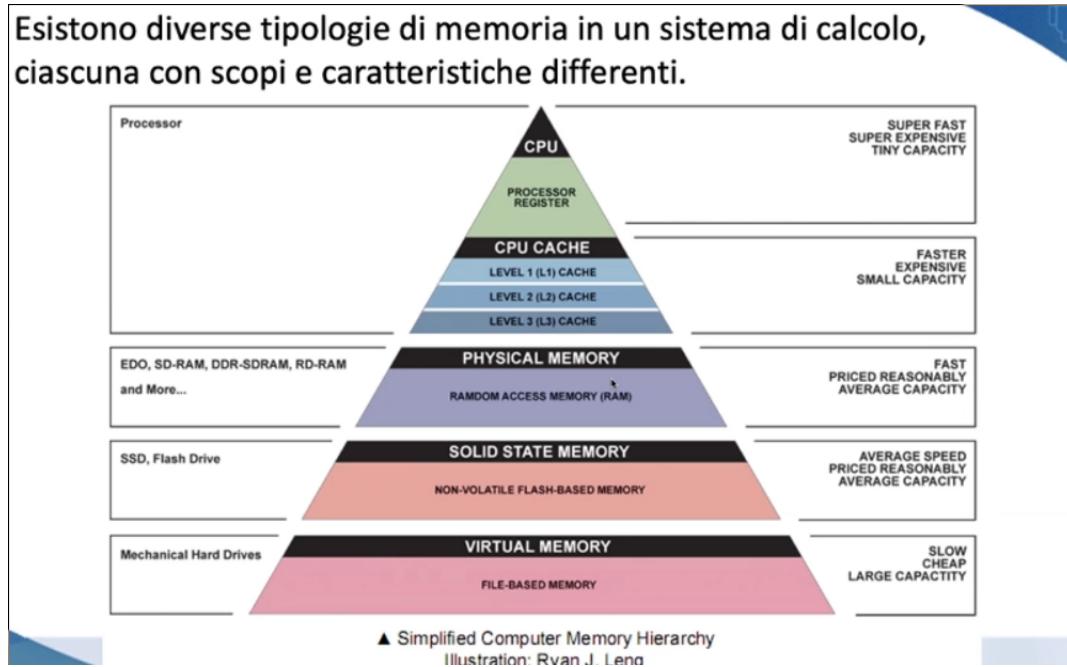
- Un programma applicativo da eseguire;
- Un interrupt → segnale generato da un controller di dispositivo o da un programma e inviato direttamente alla CPU, tramite bus;
- una exception → segnale generato da un programma a seguito di una condizione eccezionale di errore o di una specifica richiesta tramite system call.

La sequenza di avvio è:

Accensione → POST → controllo hard disk e Plug & Play → Avvio del bootloader → Avvio del SO

2.9 Tabella degli interrupt

Gli interrupt vengono gestiti mediante una strategia basata su una tabella



2.10 Memoria centrale

È rappresentata come un vettore di parole macchina dalle centinaia di migliaia all'ordine dei miliardi di parole. Nella memoria centrale vengono caricati programmi e i dati

2.11 Memoria cache

Memoria molto veloce con lo scopo di ridurre al minimo gli accessi ad unità di memoria più lente. Bisogna garantire la coerenza delle informazioni memorizzate in una o più unità di memoria cache (il cui grado di complessità aumenta a seconda del tipo di sistema)

2.12 Memoria di massa

Divise in secondarie (dischi magnetici, solidi) e terziarie (dischi ottici, CD, DVD, nastri). Le responsabilità del SO rispetto alla gestione delle memorie di massa sono:

- gestione dello spazio libero;
- assegnazione dello spazio;
- scheduling del disco.

I dati vengono scritti nelle memoria mediante file

2.13 I processi

Un processo può essere definito come un programma di elaborazione o **job**. Un processo è un entità attiva, dotata di un program counter. La sua esecuzione è sequenziale e necessita di determinate risorse:

- CPU
- memoria
- dispositivi I/O;
- file

Il SO deve garantire alcune operazioni fondamentali per i processi:

- creazione e cancellazione di processi;
- sospensione e ripristino di processi;
- sincronizzazione di processori;
- comunicazione fra processi;
- soluzioni per lo stato di **deadlock** di processi (questo compito spetta al kernel attraverso lo **scheduler**)

Gli stati di un processo

Ci si può trovare in 2 stati:

- attesa;
- pronto;

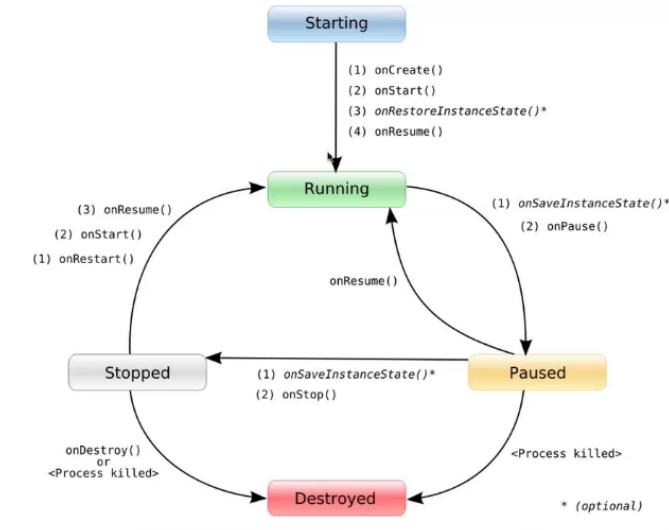
In ogni istante di tempo vi è un solo stato attivo



2.14 Activity

Una **Activity** è caratterizzata da un ciclo di vita ed in ogni istante si trova in uno dei seguenti stati:

- Active/Resumed (running) → essa è visibile e può ricevere input dall'utente;
- Paused → essa è parzialmente visibile a causa di un'altra activity in foreground, non può ricevere input dall'utente (*es* è in attesa a causa di un alert dialog presente al di sopra di essa);
- Stopped → essa non è visibile (resta in background)

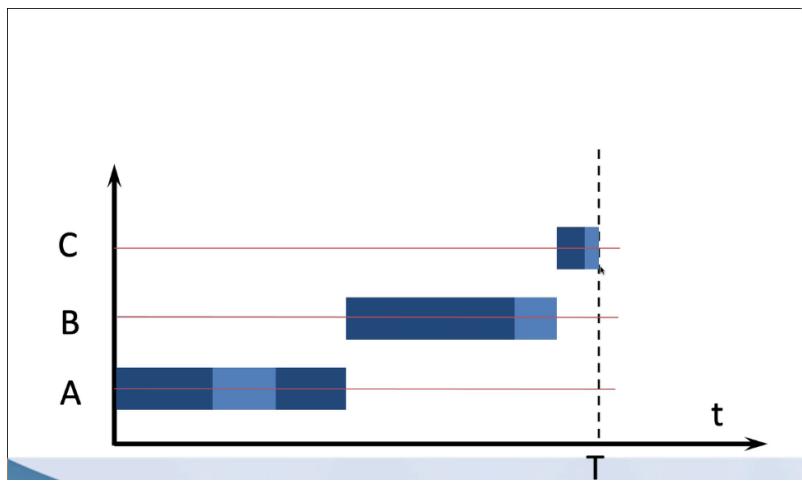


2.15 Politiche utilizzate dallo scheduler

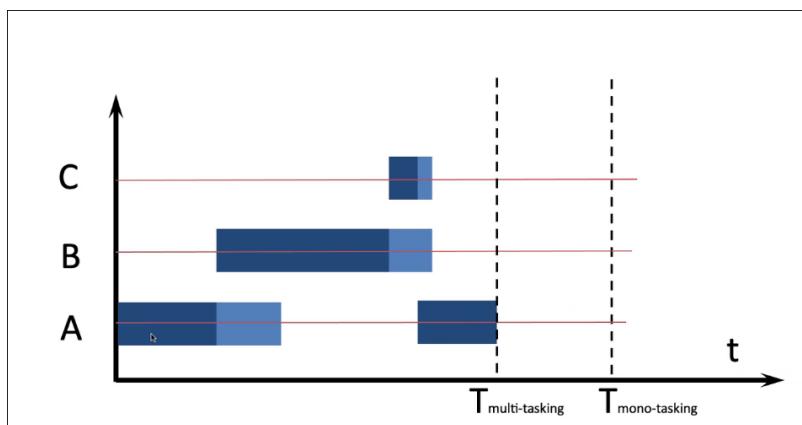
- **Preemptive** → la CPU in uso può essere tolta e passata a un altro in qualsiasi momento;
- **Non Preemptive** → una volta che il processo ha ottenuto l'uso della CPU non può essere interrotta fino a che lui stesso non la rilascia;

2.16 Sistema mono e multi tasking

La gestione della CPU (mantenere in maniera efficiente il suo uso) è un obiettivo del sistema operativo. Nel caso di un sistema mono-tasking avremo:



Nel caso di un sistema multi-tasking avremo:



(In blu scuro il tempo di utilizzo CPU e in chiaro il tempo di attesa di eventi esterni)

2.17 Throughput (utilizzo della CPU)

Si ha dal rapporto di Tp/Tt dove:

- Tp = tempo dedicato dalla CPU alla esecuzione dei processi utente;
- Tt = tempo totale di osservazione

Si ha un'alta efficienza con un throughput tendente ad 1

2.18 Overhead

Tempo impiegato dal SO per trasferire il controllo da un programma ad un altro

3 Lezione del 12-03

3.1 Sincronizzazione dei processi

Uno dei compiti che spetta al SO è la sincronizzazione dei processi. Più processi su una macchina possono essere:

- In **competizione**
 - se si cerca di usare la stessa risorsa contemporaneamente;
- In **cooperazione**
 - quando uno ha bisogno dell'altro per evolvere;

La sincronizzazione avviene grazie a variabili condivise dette **semafori** o a scambio di messaggi tra processi

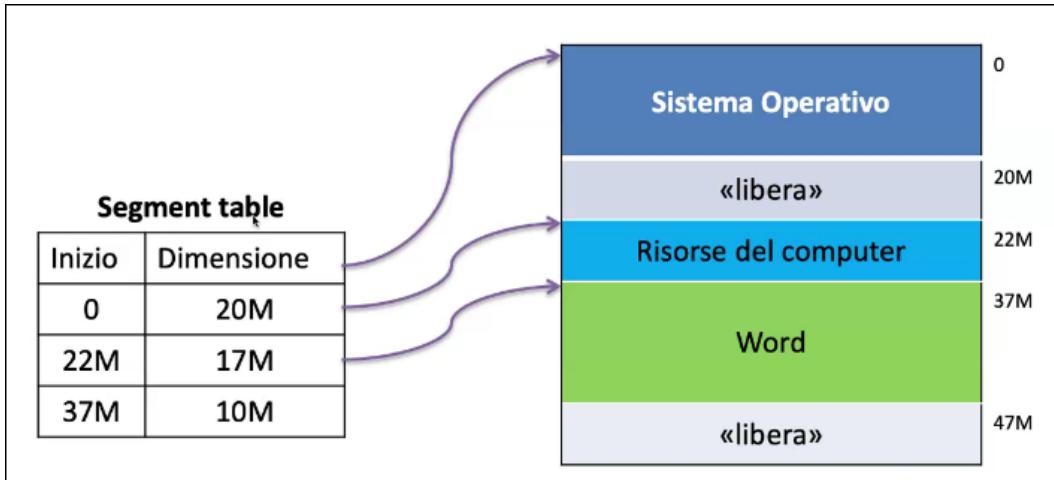
Ogni processo per essere eseguito deve essere caricato in memoria. Nei sistemi multitasking permette di caricare più processi contemporaneamente.

Il gestore della memoria deve:

- Tenere traccia di quali parti della memoria sono usate e da chi;
- Decidere quali processi caricare quando diventa disponibile spazio in memoria;
- Allocare e deallocare memoria.

Per ogni blocco di memoria il SO mantiene un **descrittore** con le informazioni essenziali. La memoria fisica è suddivisa in un determinato numero di aeree (segmenti)

Ogni entry nella **segment table** contiene l'indirizzo di base di un segmento e la sua lunghezza



Lo svantaggio principale è la frammentazione (la memoria non viene sfruttata completamente). Per ovviare a ciò esiste un processo di compattamento → traslazione dei programmi.

Politiche di gestione dei frammenti

- **First-fit** → alloca il primo frammento libero sufficiente;
- **Best-fit** → alloca il più piccolo frammento libero sufficiente;
 - ricerca sull'intera lista e produce frammenti piccoli
- **Worst-fit** → alloca il più grande frammento
 - ricerca sull'intera lista e produce frammenti grandi

Nei moderni SO si utilizza la **paginazione dei segmenti**, ossia un segmento viene realizzato tramite un insieme di pagine

La suddivisione della memoria centrale in blocchi di dimensioni fisse (pagine) tipicamente 2KB o 4KB.

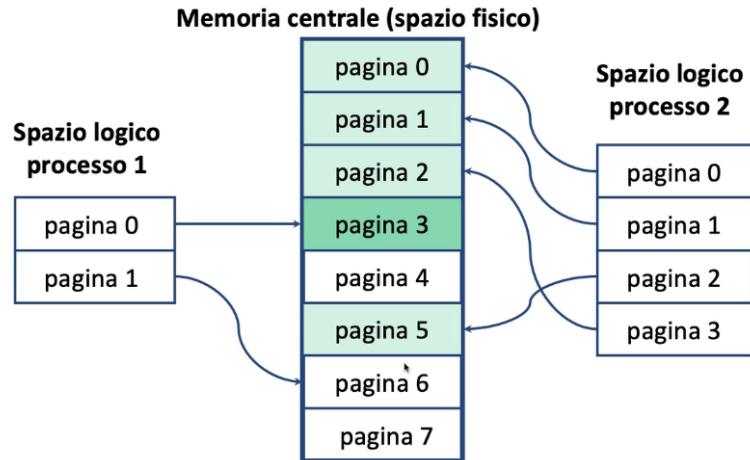
L'immagine di un processo viene suddivisa in pagine caricate all'interno della memoria principale in modo indipendente, anche in posizioni non contigue.

Il SO gestisce una **Page Table** che memorizza per ogni pagina (spazio logico) del processo e la sua posizione in memoria centrale (spazio fisico)

Frammentazione interna ed esterna

- **Frammentazione interna** → Con i blocchi di dimensioni fisse, all'interno di ogni blocco ci sarà uno spazio non utilizzato

- **Frammentazione esterna** → con i blocchi di dimensioni variabili, dopo un certo numero di allocazione e deallocazione di processi, si formeranno un certo numero di porzioni di memoria libere (buchi) di dimensioni insufficienti a contenere un processo



Se durante l'esecuzione del processo, la CPU fa riferimento ad un indirizzo che appartiene ad una pagina non caricata in memoria centrale (page fault), il SO provvede a mettere il processo in stato d'attesa e a far partire il caricamento della pagina richiesta (swap), dopodichè il processo può riprendere la sua esecuzione.

Nella paginazione esiste ancora il problema della frammentazione, che è però limitato al fatto che l'ultima pagina di ogni processo è solo parzialmente occupata. Mediamente si può considerare uno spreco pari a circa 1/2 pagina per ogni processo

3.2 Comunicazione con i dispositivi

Il gestore delle periferiche permette la comunicazione con le varie periferiche. Particolari programmi detti device driver gestiscono le operazioni di **in/out** tra le varie periferiche collegate alla macchina.

Le funzioni svolte dai device driver sono:

- Rendere trasparenti le caratteristiche fisiche tipiche di ogni dispositivo;
- Gestire la comunicazione dei segnali verso i dispositivi;
- Gestire i conflitti, nel caso in cui due o più task vogliono accedere contemporaneamente allo stesso dispositivo

3.3 File system

Gestisce le memorie di massa (organizza logicamente i dati e le possibili operazioni su di essi)

3.4 La shell dei comandi

L'interprete dei comandi o shell è quella parte più esterna di un SO che riceve ed elabora le istruzioni impartite da un utente. Il SO presenta due tipi di interfaccia:

- testuale;
- grafica (diversa tra mobile e pc);

3.5 La memoria virtuale

Sistema SW/HW in grado di simulare uno spazio di memoria centrale maggiore di quello fisicamente presente

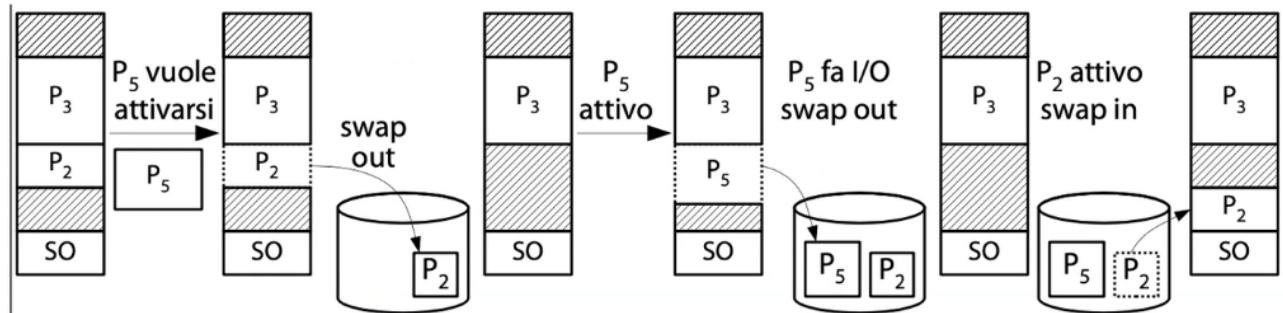
Rappresenta in modo astratto e logico la memoria centrale di un calcolatore

Memory management unit

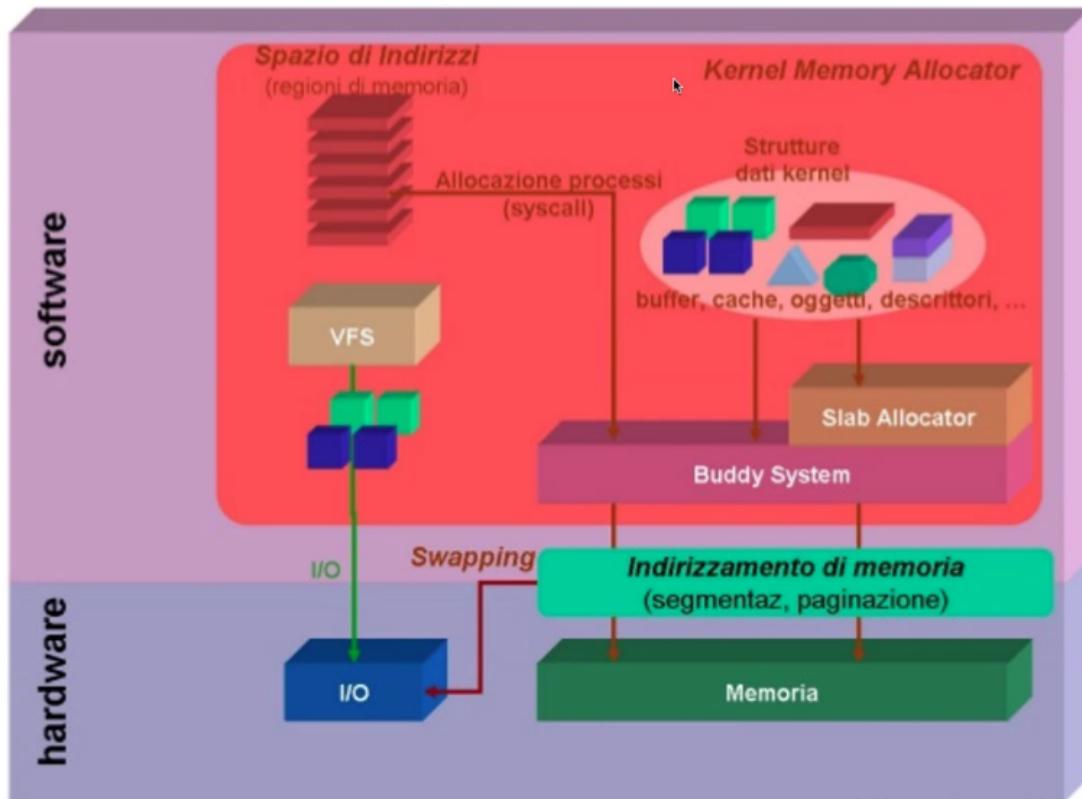
Gestisce le richieste di memoria delle applicazioni traducendole in istruzioni da impartire all'unità HW di gestione della memoria

Vantaggi:

- Esecuzione concorrente dei processi;
- Esecuzione di applicazioni con richieste di memoria maggiori della memoria fisica;
- Condivisione da parte dei processi di una singola immagine di memoria di una libreria o del programma;
- Programmi rilocati



3.6 Memoria virtuale in Linux



Buddy System → tecnica di allocazione dinamica della memoria che divide la memoria in partizioni per soddisfare una richiesta di memoria nel miglior modo possibile.

Slab allocation → tecnica di allocazione della memoria per una migliore allocazione degli oggetti. Riduce la frammentazione causata dalla allocazione e della deallocazione

Kernel Memory Allocator

È la componente del kernel che cerca di soddisfare richieste di aree di memoria provenienti dal kernel stesso e dalle applicazioni

- Il KMA risponde a richieste di memoria provenienti da:
 - altri sottosistemi del kernel per uso del SO;
 - dai programmi utenti per mezzo di system call.
- Un KMA dovrebbe avere queste caratteristiche:
 - essere veloce;
 - minimizzare sprechi di memoria;
 - ridurre la frammentazione;
 - cooperare con gli altri sottosistemi di memoria per ottenere e rilasciare pagine

3.7 Spazio di indirizzi di un processo

Lo spazio di indirizzi virtuali che ogni processo può usare è diverso dall'insieme degli indirizzi fisici che usa ad ogni istante

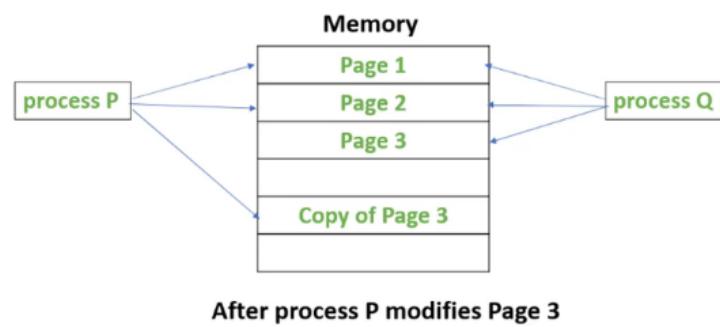
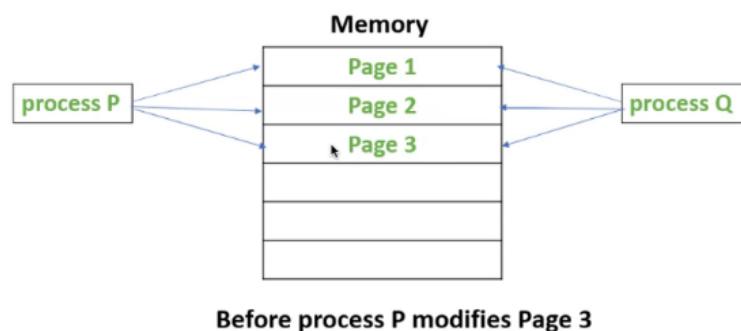
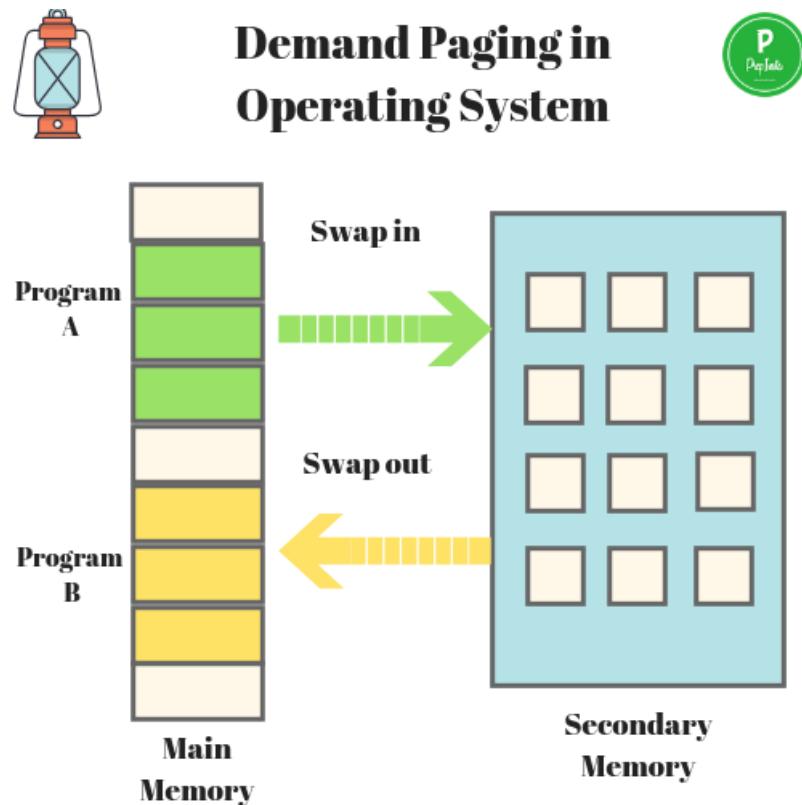
Il kernel rappresenta lo spazio di indirizzi come lista di descrittori di regioni di memoria

Alla creazione di un processo il kernel gli associa uno spazio di indirizzi che include regioni di memoria per:

- codice eseguibile;
- i dati inizializzati (e non) del programma;
- lo stack del programma;
- il codice eseguibile e i dati delle librerie condivise;
- lo heap (memoria dinamicamente richiesta dal processo);
- mapping in memoria di file usati dal programma

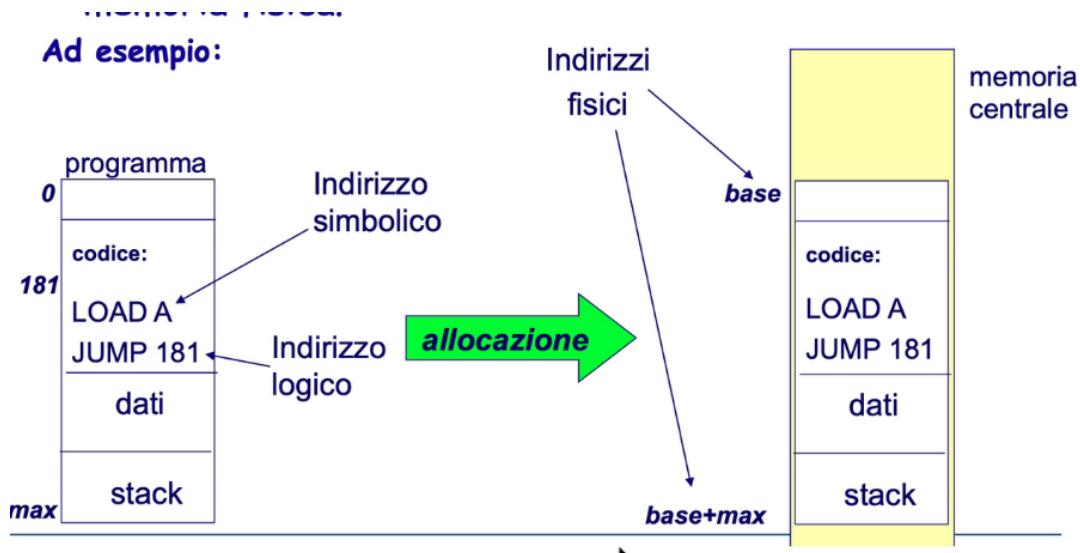
I moderni SO Unix usano strategie efficienti di allocazione di pagine basate sul [Demand Paging](#) e [Copy-on-Write](#)

Demand Paging e Copy-on-Write a confronto



Con la `fork` padre e figlio non lavorano sullo stesso spazio di indirizzamento, ma il figlio lavora su una copia dei dati del padre (a differenza dei `thread` che condividono la stessa memoria)

Ogni processo dispone di un proprio spazio di indirizzamento logico $[0, \text{max}]$ che viene allocata nella memoria fisica



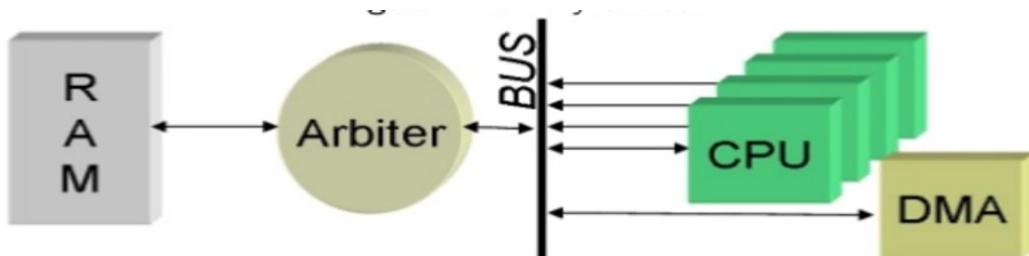
4 Lezione del 15-03

4.1 Swapping

Meccanismo HW e SW che consente di estendere la memoria centrale utilizzando spazi aggiuntivi su supporti fisici. Lo swapping è basato sulle pagine (swapped-out)

4.2 Memory Arbiter

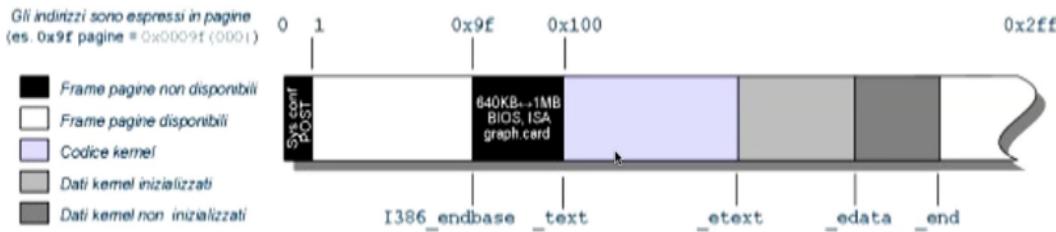
Un circuito HW che serializza (disciplina) gli accessi write-read alla ram da parte delle CPU (o dalla CPU e del DMA (accedere direttamente alla memoria interna per scambiare dati, in lettura e/o scrittura, senza coinvolgere CPU))



4.3 La RAM nei sistemi UNIX

È divisa in 2 sezioni:

- pochi mb per immagazzinare l'immagine del kernel;
- Il resto viene gestito dal sistema di memoria virtuale come memoria dinamica
 - Viene usata dal kernel per una serie di operazioni diverse



Tutto l'insieme dei meccanismi HW e SW per tradurre l'accesso ad un indirizzo di memoria virtuale (logico) nel corrispondente accesso alla locazione fisica in RAM.

Il SO non deve occuparsi di tutti gli aspetti dell'accesso alla memoria fisica (nei moderni microprocessori esiste supporto HW per la traduzione e meccanismi hHW per l'individuazione e segnalazione di errore)

4.3.1 Risultati desiderati

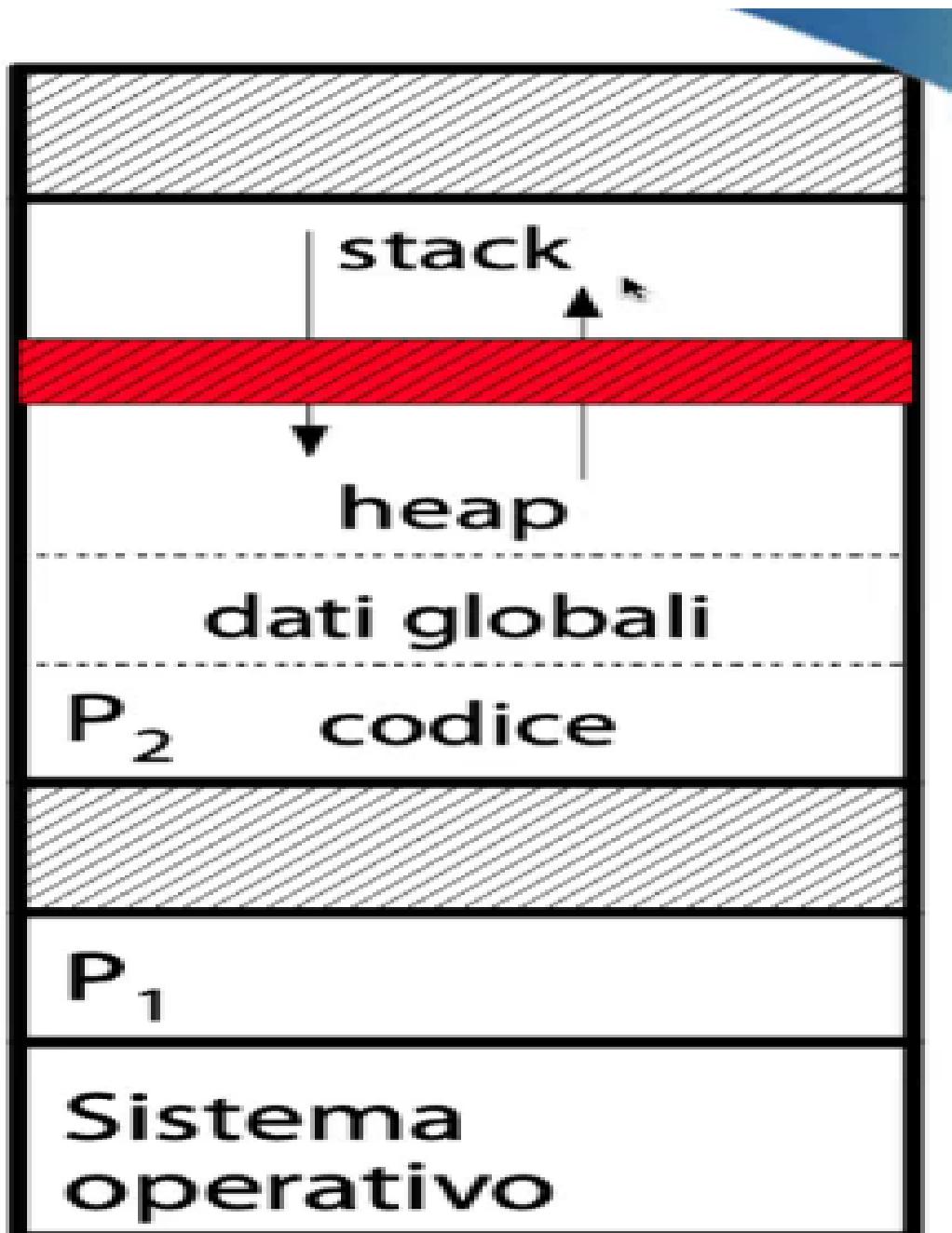
- La partizione di un processo deve essere sempre collocata in una porzione di memoria continua → swap di porzioni mirate della memoria di un processo
- Necessità di tenere lontani heap e stack
- Si deve impedire ad un programma l'accesso alla memoria di un altro programma
- Si deve impedire o limitare il buffer overflow

4.3.2 Stack & heap

Stack → heap: le strutture dati in cima all'heap contengono valori dello stack → corruzione dei dati sull'heap

Heap → stack: gli ultimi record di attivazione sono sovrascritti con valori provenienti dall'heap → il programma salta ad una locazione più o meno casuale (buffer overflow)

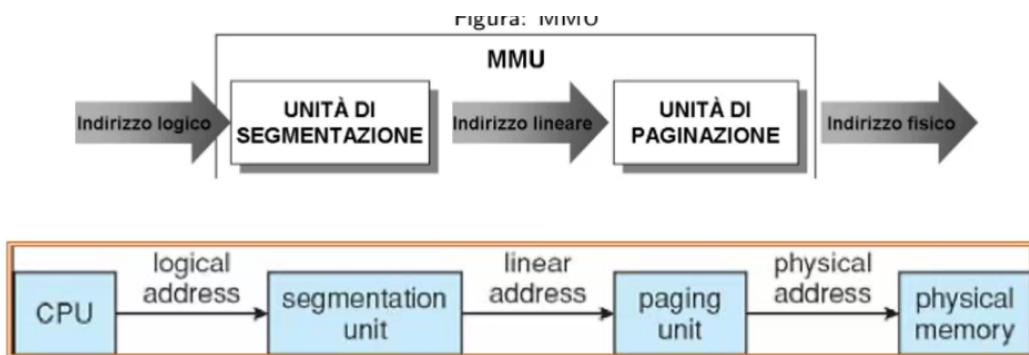
Se i dati sull'heap sono scelti bene, è possibile eseguire codice arbitrario



4.4 Memory Management Unit (MMU)

Componente della CPU che traduce indirizzi virtuali in fisici:

- Usa una circuitistica HW detta Unità di Segmentazione per tradurre indirizzi logici in indirizzi lineari
- Una una circuitistica HW detta Unità di Paginazione per tradurre indirizzi lineari in indirizzi fisici



4.5 Segmentazione

Schema di gestione della memoria che implementa una visione del programma per "blocchi funzionali"

4.6 Indirizzo logico

Un numero binario a 48 bit formato da un selettori di segmento (16 bit) e un offset (32 bit)

4.6.1 Segment selector

Identifica il segmento a cui appartiene l'indirizzo di memoria. Diviso in

- TI (1 bit) → indica in quale tabella è immagazzinato il descrittore di segmento
 - GDT → tabella dei descrittori globali;
 - LDT → tabella dei descrittori locali;
- indice (13 bit) → posizione descrittore in tabella (GDT o LDT);
- RPL (2 bit) → flag che indica il CPL (current privilege level) che aveva la CPU quando il segmento viene caricato;
- i 6 registri di segmento (cs, ss, ds, es, fs, gs) contengono i selettori di segmento
 - i rimanenti bit non sono usati (cs che ne usa 2 per il CPL)
 - * cs → code segment;
 - * ss → stack segment;

- * ds → data segment;
- * es → extra segment;
- * fs, gs → inseriti per operazioni future;



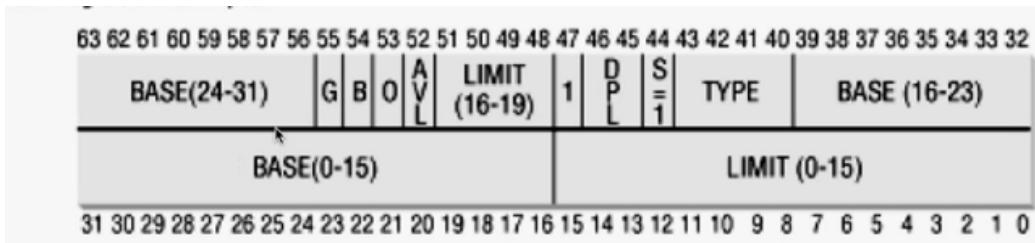
4.6.2 Offset

Indica la posizione dell'indirizzo rispetto all'inizio del segmento

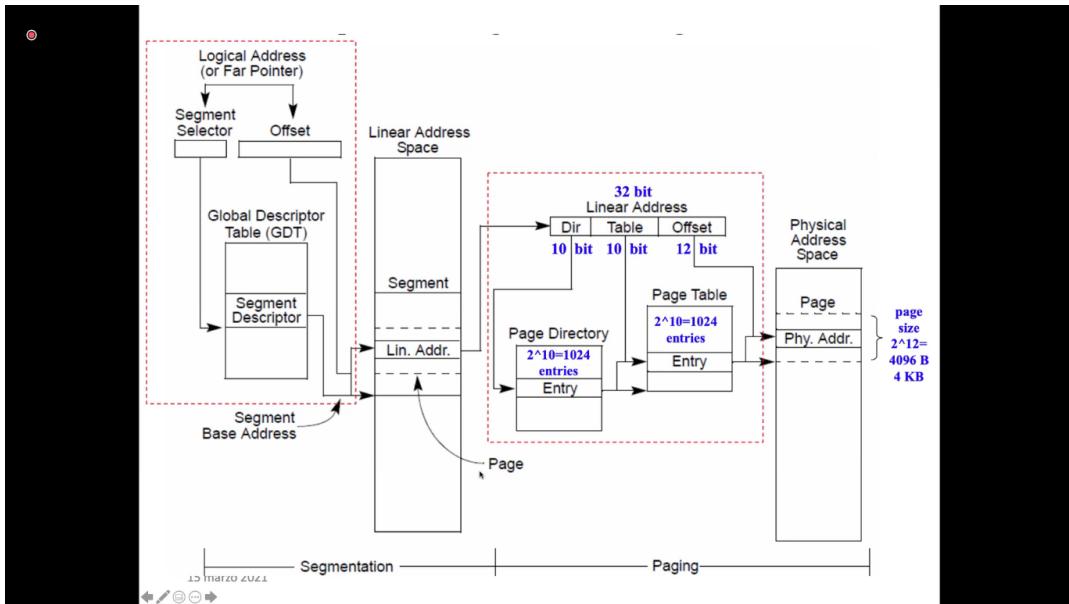
4.6.3 Descrittore di segmento

Dato di 64 bit che descrive le caratteristiche del segmento:

- BASE → indirizzo lineare base del primo byte del segmento;
- LIMIT → dimensione del segmento;
- TYPE → tipo di segmento;
- DPL (Descriptor Privile Level) → livello minimo di privilegi per accedere al segmento;
- bit 47° → 1 se presente in memoria, 0 altrimenti



4.6.4 Traduzione degli indirizzi logici in indirizzi fisici



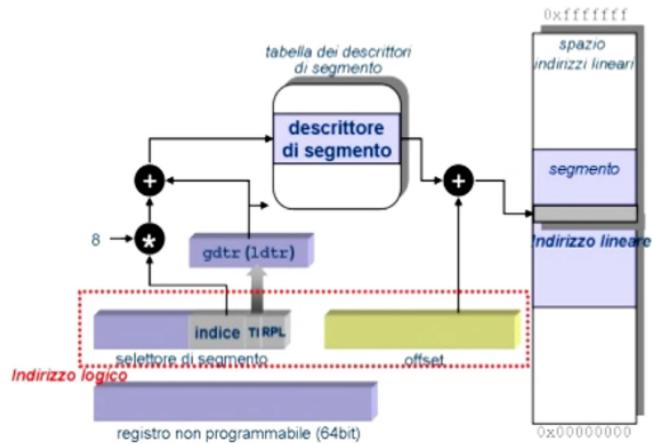
Ogni registro ha un registro non programmabile (con lo scopo di velocizzare la traduzione)

In esso è caricato il descrittore del segmento quando, per la prima volta, è caricato il selettore di segmento nel corrispondente registro segmento

Se il contenuto di un registro segmento cambia (flush), cioè un nuovo selettore è caricato, la segmentazione passa per la GTD (o LDT). Nel caso contrario (segmentazione diretta) non passa per la GTD (o LDT)

- ➊ il valore TI determina se il descrittore si trova in GDT o LDT
- ➋ l'indirizzo base della GDT (LDT) è nel registro gdtr (ldtr)
- ➌ l'indice $\times 8 + \text{ind.base} = \text{posiz. descr. in GDT (o LDT)}$. Copia descr. in r.n.p.
- ➍ dal descrittore si ricava l'indirizzo (lineare) base del segmento
- ➎ ind. base segmento + offset = indirizzo lineare tradotto

Figura: segmentazione: accesso in GDT

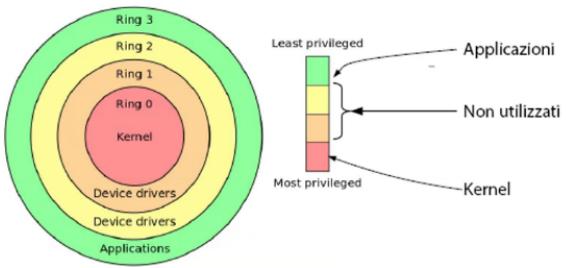


4.6.5 Privilegi di esecuzione

Meccanismo di difesa da istruzioni o accessi alla memoria causati da malfunzionamenti HW o comportamenti malevoli

La CPU è dotata di 4 modi di utilizzo disposti ad anello

- Ring 3:** applicazioni utente.
- Ring 2:** device driver.
- Ring 1:** device driver, hypervisor.
- Ring 0:** device driver e basso livello SO (kernel).

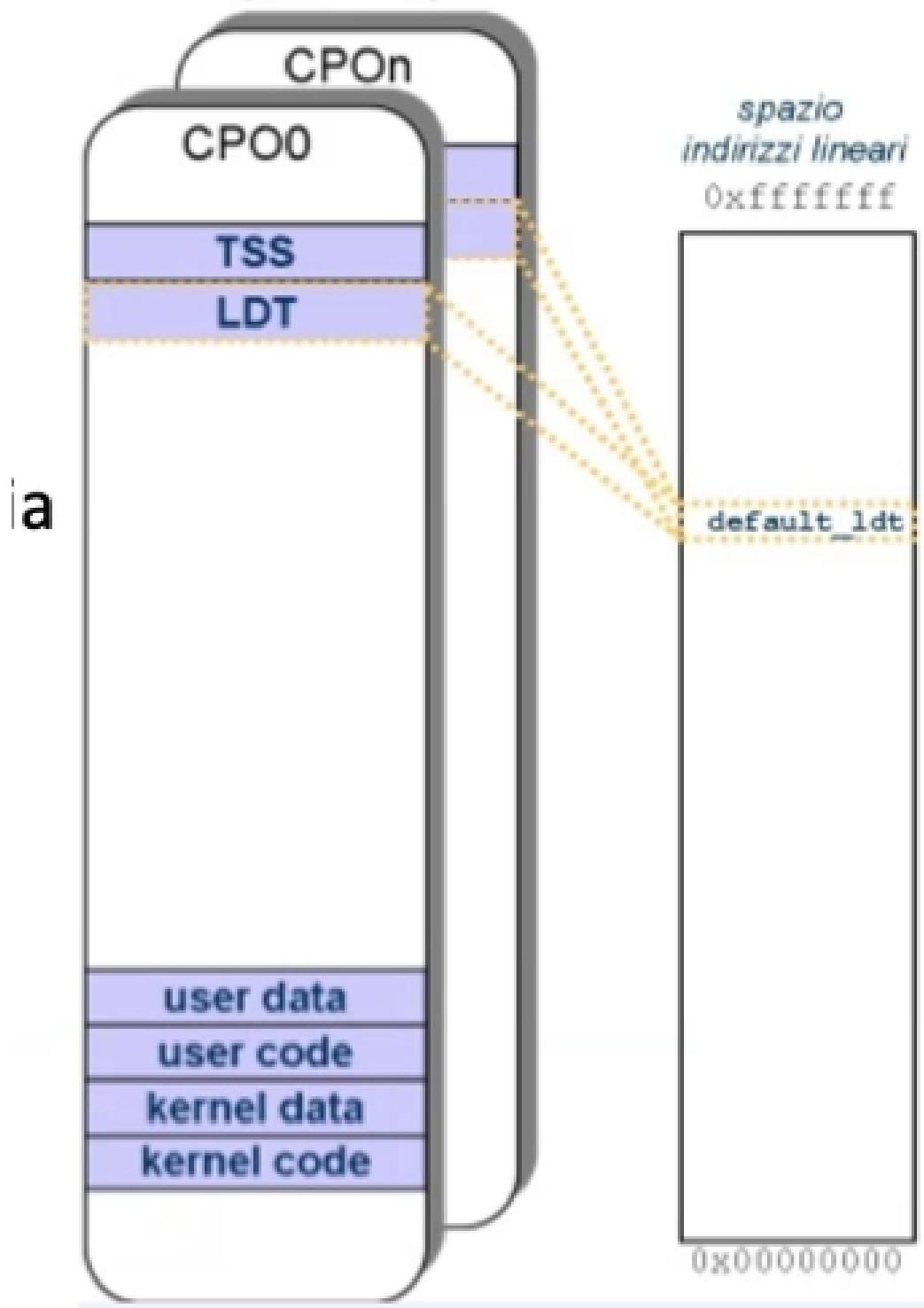


4.6.6 Tabelle dei Descrittori di Segmento

Vengono immagazzinati in GDT o LDT. Queste tabelle sono immagazzinate in memoria RAM.

I registri gdtr e ldtr contengono rispettivamente gli indirizzi fisici base della GDT e LDT

Figura: segmenti in GDT



4.7 Segmentazione in sistemi LINUX

Linux usa principalmente la paginazione e in modo limitato la segmentazione (si limita il numero di flush)

4.7.1 I segmenti in Linux

1. Kernel code Segment Corrisponde ad un segmento di codice del kernel ed il suo segment selector è definito con la macro `__KERNEL_CS`
2. Kernel Data Segment Corrisponde ad un segmento di dati del kernel ed il suo segment selector è definito con la macro `__KERNEL_DS`
3. User Code Segment Corrisponde ad un segmento di codice condiviso da tutti i processi utente ed il suo segment selector è definito con la macro `__USER_CS`
4. User Data Segment Corrisponde ad un segmento di dati condiviso da tutti i processi utente ed il suo segment selector è definito con la macro `__USER_DS`

4.7.2 Spazio degli indirizzi lineari

Intervallo di indirizzi lineari da 0 (0x00000000) a 4G (0xFFFFFFFF).

La dimensione massima dello spazio di indirizzi è determinata dall'architettura della CPU

4.7.3 Pagina di indirizzi lineari

Intervallo di indirizzi lineari di dimensione fissa (in Linux 4K); l'indirizzo base dell'intervallo è multiplo della dimensione.

4.7.4 Segmentazione vs Paginazione

Entrambi sono ridondanti → entrambi **partizionano/assegnano** la memoria fisica tra processi.

- La segmentazione assegna intervalli di indirizzi lineari a diversi ai processi;
- La paginazione mappa un intervallo di indirizzi lineari su intervalli di indirizzi fisici diversi

4.7.5 Page

Un blocco di dati che può essere scritto su un frame pagina.

Questo termine si riferisce sia alla pagina di indirizzo che alla pagina di dati memorizzati in RAM in corrispondenza.

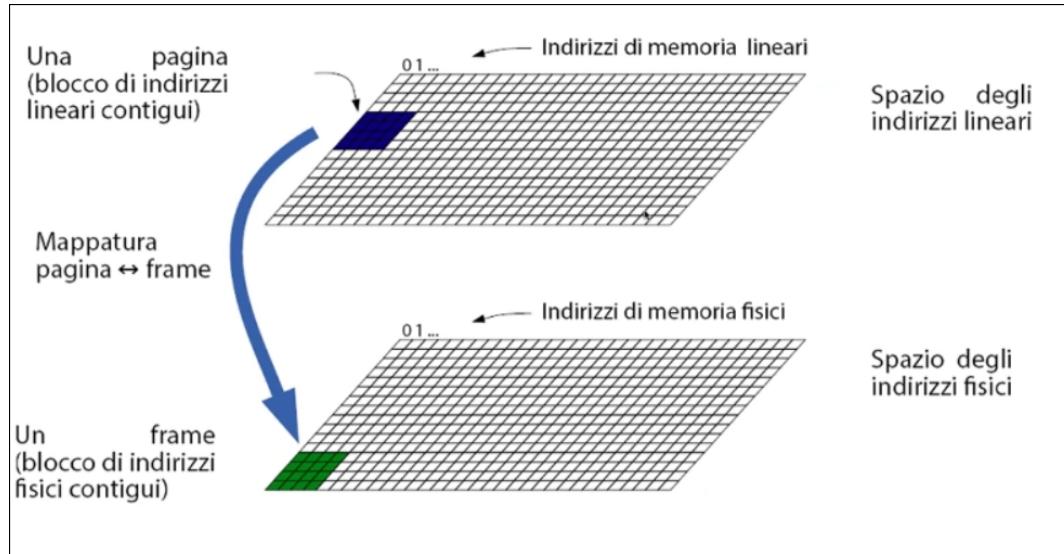
Una frame pagina memorizza esattamente il blocco di dati (o istruzioni) contenuti in una pagina (di indirizzi lineari).

La paginazione mappa pagine di indirizzi lineari su frame pagine; una pagina è in RAM o in disco

1. Page Frame Area fisica (intervallo di locazioni contigue) di memoria RAM di dimensione fissata
 2. Paging Unit Circuitistica della MMU che converte indirizzi lineari in fisici. Nella stessa pagina, indirizzi lineari contigui sono paginati su indirizzi fisici contigui.
- L'ultimo indirizzo di una pagina e quello della pagina successiva possono essere mappati su indirizzi fisici non contigui

I permessi di accesso e i privilegi associati ad una pagina valgono per tutti gli indirizzi in essa contenuta.

La paginazione verifica anche che la richiesta di accesso abbia privilegi e modalità compatibili con quelli associati alla pagina di appartenenza



3. La tabella delle pagine Ha una dimensione alta (viene mantenuta in RAM).

Nei processori intel l'indirizzo fisico iniziale della tabella è salvato nel registro di controllo CR3.

Come detto essendo gigantesca la sua dimensione si fa uso di uno **schema di paginazione gerarchico** → ciascun elemento della tabella delle pagine punta all'inizio di una tabella più piccola (che puntano ai frame fisici)

4.7.6 Tipi di Paginazione

- A 2 livelli (sistemi a 32 bit);
- A 3 livelli (in quanto il precedente non è più sufficiente);
- A 4 livelli (usati dai sistemi Linux), con 4 tipi di tabelle
 - Page Global Dir (PGD);
 - Page Upper Dir (PUD);
 - Page Middle Dir (PMD);
 - Page Table (PT).

4.7.7 TLB

Cache HW che velocizza la traduzione di indirizzi lineari in fisici

La prima volta che un indirizzo lineare è tradotto, l'indirizzo fisico corrispondente è registrato in un elemento della TLB. Per usi successivi dello stesso indirizzo lineare si userà l'indirizzo fisico presente in TLB.

Ogni CPU ha la propria TLB → sincronizzazione non necessaria:

- Processi che eseguono su diverse CPU possono usare stessi indirizzi logici corrispondenti a indirizzi fisici diversi;
- TLB funziona da cache per gli elementi della Page Table;
- Se cambia l'indirizzo fisico di una frame pagina (swapping), l'elemento della TLB corrispondente è invalidato;
- Se cambia il valore del registro CR3 (PD) tutta la TLB viene invalidata

5 Lezione del 19-03

5.1 La programmazione di sistema

Consiste nell'utilizzare l'interfaccia di system call fra il kernel e le applicazioni che girano sotto UNIX.

5.1.1 System call

Per i programmi C, le system call sono come funzioni. Ogni system call ha un prototipo:

- `pid_t` (0 se padre, altro valore se figlio)

Tra le system call ricordiamo:

- `getpid`;
- `getppid`;
- `getgrp`;

Che forniscono degli attributi dei processi (PID, PPID, gruppo etc)

- `fork` → crea un processo figlio duplicando il chiamante;
- `exec` → trasforma un processo sostituendo un nuovo programma nello spazio di memoria del chiamante;
- `wait` → permette la sincronizzazione dei processi;
- `exit`;

5.1.2 Fork

Il prototipo è `pid_t fork()`, dove:

- `pid_t` è definito negli include di sistema.

Il valore restituito da `fork` può essere usato per distinguere tra processi genitore e processo, infatti al genitore viene restituito il PID del figlio, al figlio viene invece restituito 0. Restituisce 0 al figlio e pid al padre, oppure -1 (solo al padre) in caso di fallimento

5.1.3 Processo Zombie

Un processo diviene zombie dopo essere terminato e prima che il processo padre abbia eseguito una `wait` su quel figlio

5.1.4 Esempio

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(){
    pid_t pid;
    int status;
    pid = fork();
    if (pid)
        sleep(2);
    pid = wait (&status);
    if (WIFEXITED (status)) {
        printf ("stato %d\n", WEXITSTATUS(status));
    } else{
        printf ("Processo %d, figlio. \n ", getpid());
        _exit(17);
    }
}
```

5.2 Gestione dei processi in Linux

Si definisce processo una qualsiasi istanza di programma in esecuzione.

Supporto di multithreading, attraverso i lightweight process (queste condividono le risorse) → in questo modo vengono supportati i thread.

5.3 Process Descriptor

Il kernel monitora costantemente l'attività dei processi mediante una struttura di tipo `task_struct` che prende il nome di Process Descriptor

Questo descrittore contiene una serie di puntatori a strutture secondarie (che contengono le relative informazioni)

Il kernel contiene la lista dei processi in una lista doppiamente linkata chiamata task list

5.4 Stati da un processo

- `TASK_RUNNING`;
- `TASK_INTERRUPTIBLE`;
- `TASK_UNINTERRUPTIBLE`;
- `TASK_STOPPED`;
- `TASK_TRACED`;
- `EXIT_ZOMBIE`;
- `EXIT_DEAD`.

5.5 Identità di un processo

Qualunque processo in Linux è caratterizzato da un process descriptor ed il puntatore a tale struttura prende il nome di process descriptor pointer. Linux identifica i processi assegnando loro un Process ID (PID) che viene memorizzato nel campo `pid_field` del process descriptor.

I PID vengono assegnati in ordine crescente e riutilizzati solo dopo aver raggiunto il valore massimo. Linux tiene traccia dei PID nuovamente disponibili in un `pidmap_array`

5.6 Relazioni fra processi

I principali sono:

- Relazione `padre/figlio`;
- Relazione processi fratelli.

Ma ne esistono molti altri in base al condizione che andiamo a verificare.

5.7 Le liste di processi RUNNING

Nelle precedenti versioni di linux, la selezione del processo da eseguire avveniva scorrendo una *runqueue* alla ricerca del processo con priorità più alta.

Le attuali versioni implementano un algoritmo di selezione in tempo costante, mantenendo **k liste** di processi, dove k rappresenta la priorità (un valore intero tra 0 e 139)

5.8 I file hash

Si servono di una funzione di hash o funzione di randomizzazione, tale che applicata al valore di uno dei campi restituisce l'indirizzo del blocco in cui è memorizzato il record. Consentono operazioni di ricerca molto efficienti.

Per file interni, l'hash è implementato con una tabella di hash.

5.8.1 Funzione di accesso h

Definita come:

$$h:K \rightarrow \{0,1,2,\dots, m-1\}$$

Dove:

- $K \rightarrow$ insieme dei valori distinti che il campo chiave può assumere;
- $m \rightarrow$ dimensione del vettore in cui si intende memorizzare gli elementi della tabellla

5.8.2 Divisione

Definita come:

$$h(k) = k \bmod |T|$$

Presenta una bassa complessità, ma può generare fenomeni di agglomerazione

5.8.3 Risoluzione delle Collisioni

Esistono diverse strategie per la risoluzione delle collisioni:

- Concatenamento → alla i-esima posizione della tabella è associata la lista degli elementi tali che $h(k)=i$;
- Indirizzamento aperto → tutti gli elementi sono contenuti nella tabella.

La scelta della strategia influisce sull'efficienza delle operazioni di inserimento, ricerca e cancellazione.

5.8.4 Tabella Hash per la runqueue

Il Kernel mantiene quattro tabelle hash, che gli permettono di collegare il PID di un processo al suo **process descriptor**

Le collisioni sono gestite mediante liste concatenate

5.9 Organizzazione dei processi

La runqueue contiene solo i processi nello stato `TASK_RUNNING`.

I processi nello stato `TASK_STOPPED` `EXIT_DEAD` `EXIT_ZOMBIE` non vengono raggruppati in nessuna lista.

I processi che si trovano nello stato di `TASK_INTERRUPTIBLE` e `TASK_UNINTERRUPTIBLE` vengono raggruppati a seconda dell'evento per cui sono stati accodati

Poichè lo stato di un processo non è sufficiente a decidere la politica di risveglio da adottare si fa uso delle wait queue

5.9.1 Wait queue

Usate dal kernel per gestire diverse situazioni:

- Gestione degli interrupt;
- Sincronizzazione dei processi;
- Timing.

Sono implementate mediante liste doppiamente concatenate, i cui elementi puntano a process descriptors. Sono fornite inoltre di uno spinlock poichè possono essere utilizzate sia delle principali funzioni del kernel, che dai gestori degli interrupt

5.9.2 Struttura di una Wait queue

Il campo `task_list` contiene il puntatore alla lista di processi che attendono nella coda. La decisione di risvegliare un processo in attesa in una coda dipende da diversi fattori. In particolare bisogna valutare se esso attende per una risorsa che può essere acceduta in modo esclusivo.

Esistono due tipi di processi sleeping:

- Esclusivi → identificati dal campo flags impostato a 1;
- Non esclusivi → identificati dal campo flags impostato a 0;

5.10 Duplicazione delle risorse

I SO Unix si basano sulla generazione a catene di processi per servire le richieste dell'utente. Unix tratta i processi allo stesso modo.

Le risorse di un processo sono duplicate e assegnate ai figli. Questo modalità è inefficiente.

Nei moderni sistemi Linux questo problema viene risolto attraverso diverse tecniche:

- Copy-or-Write;
- Lightweight processes (threads);

5.10.1 Chiamate di sistema in Linux

Ne esistono 3:

- fork() → valore di ritorno
 - nel contesto di esecuzione del padre il PID del processo figlio;
 - nel contesto di esecuzione del figlio ritorna 0;
 - padre e figlio non condividono lo spazio di indirizzi
- vfork() → analogo al fork;
- clone() → valore di ritorno
 - PID del processo creato;
 - -1 in caso di fallimento.
 - padre e figlio condividono le risorse indicate

Tutte e 3 invocano la stessa funzione di servizio `do_fork()`

5.11 Bomba fork

Attacco di tipo denial of service contro un computer che utilizza la funzione fork.

Es in bash

```
:() { : | : & };:
```

5.12 Terminazione di un processo

Un task termina su sua richiesta (sincrono) quando:

- finisce di eseguire il proprio codice
 - chiamata esplicita o implicita della funzione di sistema `exit()` (o `pthread_exit()`)

Un task non termina su sua richiesta (asincrono) quando:

- riceve segnali di terminazione;
- avviene un'eccezione.

In tutti i casi è il Kernel che ne forza la terminazione:

- rilasciando tutte le risorse del task;
- rimuovendo il task dalle liste del sistema e deallocandole il descrittore + KMS

5.13 I thread

Rappresenta uno dei possibili sottoprocessi eseguibili all'interno di esso. Sono quindi parti di un processo che vengono eseguite in maniera concorrente ed asincrona

Nei thread vengono condivise le medesime informazioni di stato, la memoria ed altre risorse di sistema

La differenza è il meccanismo di attivazione:

- l'operazione di attivazione di un processo è un'operazione onerosa;
- l'operazione di attivazione di un thread richiede meno risorse.

5.13.1 Kernel-level e user-level thread

Thread a livello utente

Sono gestiti come uno strato separato sopra il nucleo del SO, e realizzati tramite una libreria di funzioni per la creazione, lo scheduling, e la gestione, senza alcun intervento diretto del nucleo

Thread a livello del nucleo

Gestiti direttamente dal SO

Distinguiamo 3 tipi di modelli:

1. Modello da molti a uno Fa corrispondere molti thread a livello utente a un singolo thread al livello del nucleo
2. Modello da uno a uno Fa corrispondere ciascun thread a livello utente con un thread al livello del nucleo.
3. Modello da molti a molti Fa corrispondere più thread a livello utente con un numero minore o uguale di thread al livello del nucleo

5.13.2 Clone()

Esiste la possibilità attraverso la system call clone() di generare thread. Questa accetta come parametro un insieme di flag che definiscono quante e quali risorse del processo genitore debbano essere condivise dal processo figlio, come:

- CLONE_FS → le informazioni sul file system sono condivise;
- CLONE_VM → condivisione dello stesso spazio di memoria;
- CLONE_SIGHAND → condivisione dei gestori dei segnali;
- CLONE_FILES → condivisione dell'insieme dei file aperti

5.13.3 Pthreads

Lo standard POSIX (Portable Operating System Interface for UniX) → definisce l'API per la creazione e la sincronizzazione dei thread.

Non si tratta di una realizzazione ma di una definizione del comportamento dei thread.

I progettisti di SO possono realizzare le API così definite come meglio credono

6 Lezione del 22-03

6.1 Kernel thread

Linux delega alcune attività critiche e periodiche a task, detti **kernel threads**:

- Flush di cache disco;
- Swapping out di pagine non utilizzate;
- Servizi di connessione rete;
- ...

I kernel threads:

- Eseguono solo in kernel mode;
- Sono schedulati in modo intermittente;
- Eseguono una funzione del kernel specifica.

La funzione `kernel_thread()` ha il compito di creare un nuovo kernel thread. Riceve come parametri la funzione del kernel da eseguire e i suoi comandi.

6.1.1 Processi

- Processo 0 (swapper) → alloca diverse strutture dati referenziate dai processi;
- Processo 1 (initi) → creato da swapper, è l'antenato di tutti i processi;
- kswapd → recupera memoria;
- ksoftirqd → esegue tasklet di interruzioni;
- pdflush → esegue flush di buffer dirty su disco.

6.1.2 Process switch

Si intende l'attività del kernel di sospendere l'esecuzione di un processo per sostituirla con quella di un processo differente.

Viene effettuato sempre in Kernel mode.

Gli elementi coinvolti sono 2:

- Contesto HW;
- politiche di avvicendamento dei processi.

Contesto SW

Occorre ripristinare:

- Lo stato dei registri della CPU;
- risorse utilizzate;
- stato della memoria

Alcune informazioni sono salvate in memoria (`task_struct`), altre nella CPU (`registri`).

Contesto HW

Conservato in:

- process descriptor del processo;
- Kernel Model stack del processo.

6.2 Scheduling dei processi

Linux è un sistema multi-tasking e time-sharing.

Gli aspetti importanti legati allo scheduling sono 3:

- Politiche di scheduling;
- Algoritmi di scheduling;
- Strutture dati e le funzioni che implementano lo scheduling;

6.2.1 Scheduling policy

- Tempo di risposta breve;
- elevato throughput per i processi in background;
- evitare la starvation di processi;
- condivisione di risorse fra processi ad alta e bassi priorità.

In linux il tempo di esecuzione è suddiviso in *slice* → al termine di questo è soggetto ad un process switch.

Diversi modi di categorizzazione dei processi:

- I/O bound;
- CPU bound.

Possono essere classificati in:

- interattivi → richiedono interazioni con l'utente;
- batch → non richiedono interazioni con l'utente e sono spesso eseguiti in modalità background;
- Real-time → vincoli di scheduling molto stretti e non possono essere bloccati da processi con priorità inferiore.

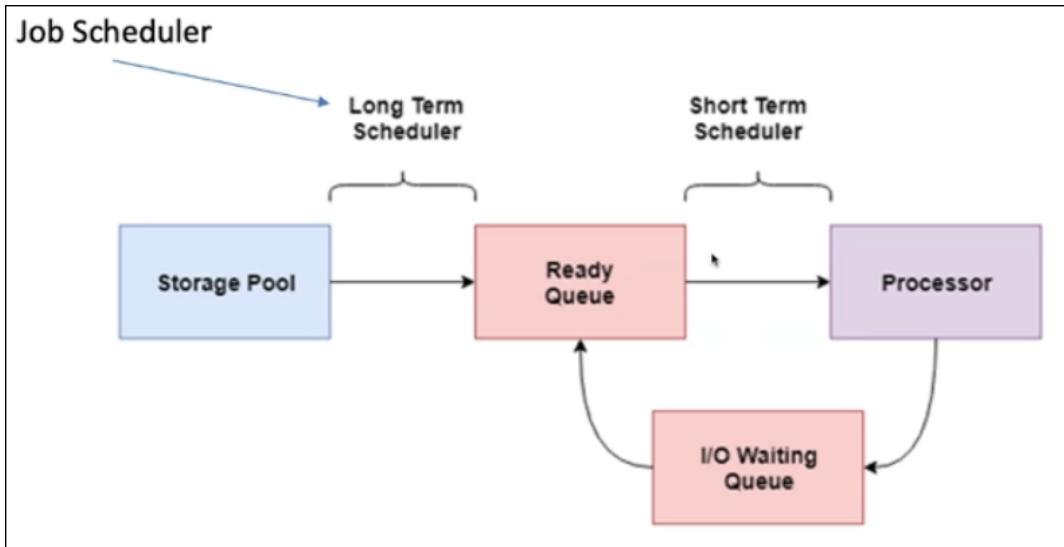
6.2.2 Single e multi-core

Il passaggio da single a multi-core è avvenuto quando ci si è resi conto che non conviene aumentare ulteriormente la frequenza di un core unico a causa dell'eccessivo consumo (fino a 100W) e calore dissipato. Il multi-core aggira questo problema, ma richiede politichè ed algoritmo di scheduling in grado di sfruttare al meglio tale possibilità

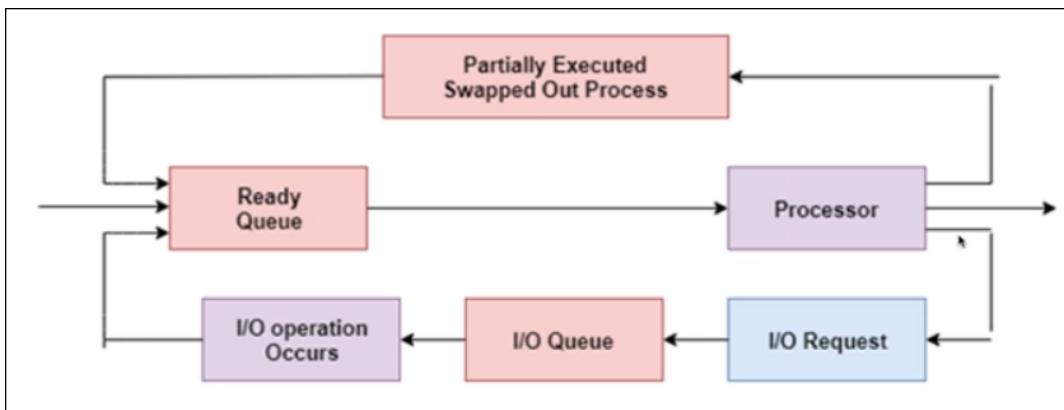
6.2.3 Tipologie di Scheduling

- Long-Term scheduling → un nuovo processo viene aggiunto all'insieme dei processi che devono essere eseguiti;
- Middle-Term scheduling → un nuovo processo viene aggiunto all'insieme dei processi che sono parzialmente o completamente in memoria;
- Short-Term scheduling → seleziona direttamente un processo disponibile da sottomettere ad un processore.

Long-Term e Short-Term scheduling



Middle-Term scheduling



6.2.4 Caratteristiche di uno scheduler

- Trasparente al programmatore;

- Minimizzare i tempi di context switch;
- Bilanciare il carico in maniera equa;
- Assicurare che se vi sono task con priorità maggiore, questi vengano eseguiti prima degli altri;
- I processi interattivi devono rispondere velocemente alle richieste da parte dell'utente;
- Ai job che lavorano in background deve comunque essere garantita la possibilità di portare a termine il loro lavoro in tempo ragionevoli;
- Nessun processo dovrebbe soffrire di starvation.

6.2.5 Prelazione

Un processo può essere prelazionato se:

- nella run-queue entra un processo con priorità maggiore di quello corrente;
- esso ha esaurito il suo quanto di tempo

Un processo prelazionato è diverso da un processo sospeso → resta in uno stato di `TASK_RUNNING`.

Critico per sistemi a prelazione è la durata del quanto di tempo:

- Se il quanto è troppo breve, i process switch sono molto frequenti causando un elevato overhead;
- Se il quanto è troppo lungo, non si ha più la percezione da parte dell'utente che i processi siano eseguiti in modo concorrente.

La scelta della durata del quanto è un compromesso. Linux sceglie la più lunga durata che non degradi le prestazioni del sistema.

6.2.6 Algoritmo di scheduling

Basata sulle code di priorità e garantisce prestazioni $O(1)$ sia al crescere del numero di processi, che del numero di CPU (ciascuna presenta la propria coda).

Lo scheduler ha sempre almeno un processo in stato `TASK_RUNNING` da selezionare: lo *swapper*.

1. `SCED_FIFO`

Quando lo scheduler assegna ad una CPU, lascia il process descriptor nella sua posizione corrente all'intero della lista della run-queue (Realtime FIFO scheduler).

Il sistema non interromperà un processo in esecuzione con FIFO ad eccezione dei seguenti casi:

- Un altro processo FIFO con priorità maggiore diventa pronto;

- Il processo è bloccato da una I/O;
- Il processo rilascia volontariamente il processore.

Quando un processo viene interrotto, questo viene inserito nella coda associata alla sua priorità. Nel caso in cui più di un processo ha priorità alta, viene scelto il processo che ha atteso di più

2. SCHED_RR

Quando lo scheduler assegna il processo ad una CPU, pone il process descriptor alla fine della lista della run-queue. (Realtime RR scheduler). A differenza di quelle FIFO nello SCHED_RR viene aggiunta di un quanto di tempo associato ad ogni thread

3. SCHED_OTHER

Un processo time-shared convenzionale (Round Robin scheduling policy)

6.2.7 Scheduling di processi convenzionali

In Linux ogni processo ha una propria priorità statica. Il Kernel riserva 40 valori per rappresentare questa priorità (da 100 a 139) → all'aumentare del valore diminuisce il livello di priorità (I livelli da 0 a 99 sono riservate ai processi Real Time).

Un nuovo processo eredita sempre la priorità statica del padre, tuttavia un utente può cambiare il nice del processo attraverso le chiamate a sistema *nice()* e *setpriority()*

6.2.8 Quanto temporale di base

Dalla priorità statica lo scheduler calcola il **base time quantum**, ossia la durata del quanto temporale di esecuzione di un processo. Terminato quest'ultimo il processo viene fermato a favore dei processi concorrenti che possono eseguire.

A ciascun processo vengono assegnati quanti temporali di lunghezza variabile a seconda delle caratteristiche del processo. Il calcolo del nuovo quanto viene effettuato con la seguente equazione:

$$\text{quanto temporale}_{\text{ms}} = (140 - \text{priorità statica}) \times 20 \text{ se la priorità statica} < 120$$

$$\text{quanto temporale}_{\text{ms}} = (140 - \text{priorità statica}) \times 5 \text{ se la priorità statica} \geq 120$$

6.2.9 Durata del quanto temporale di base

Ogni processo è caratterizzato anche da una priorità dinamica che va da 100 (priorità massima) a 139 (priorità minima). Questa priorità è quell'attributo considerato dallo scheduler nel momento in cui deve trovare il processo più adatto da eseguire.

$$\text{priorità dinamica} = \max(100, \min(\text{priorità statica} - \text{bonus} + 5, 139))$$

Il bonus è una variabile compresa tra 0 e 10. Il valore che lo scheduler attribuisce a questa variabile → dipende dal valore assunto dal tempo medio di sleep (average sleep time)

Il rovescio della medaglia è la conversione di processi ad altra priorità, che nel caso di alti sleep time rischiano di diventare processi interattivi

6.2.10 Average sleep time

Tempo medio che il processo passa nello stato sleep. Non viene calcolato effettuando banalmente la media degli intervalli di tempo:

- Il tempo speso in sleep quando il processo è in `TASK_INTERRUPTIBLE` o `TASK_UNINTERRUPTIBLE` pesa diversamente;
- Diminuisce quando un processo è in esecuzione;
- Non può superare il valore di 1 secondo.

Un processo viene considerato interattivo se:

$$\text{priorità dinamica} \leq 3 \times \frac{\text{priorità statica}}{4+28}$$

o in maniera equivalente

$$\text{bonus -5} \geq \frac{\text{priorità statica}}{4+28}$$

7 Lezione del 26-03

7.1 Processi batch e interattivi

La natura di un processo rimane piuttosto invariata. Se effettivamente lo scheduler scegliesse sempre il processo migliore, si penalizzerebbe l'esecuzione dei processi a bassa priorità.

Per risolvere questa problematica il kernel attua due policy:

- processi attivi (*active processes*)
 - processi che non hanno ancora completato il loro quanto temporale
- processi scaduti
 - processi che hanno appena terminato il loro quanto temporale e finché non si esauriscono i processi attivi, questi continueranno a non eseguire

Un processo batch se ha finito il suo quanto temporale viene sempre aggiunto alla lista di processi scaduti. Questo non è vero per un processo interattivo, ricalco del suo quanto temporale → resta nella lista dei processi attivi a meno che:

- il primo dei processi della lista expired ha aspettato oltre una determinata soglia;
- un processo scaduto ha priorità statica maggiore del processo interattivo
 - in questo caso lo scheduler è programmato per aggiungere alla lista dei processi scaduti

7.2 Scheduling dei processi Real-time

Priorità da 0 a 99. Questo tipo di processi sono sempre attivi. Lo scheduler scegli quello che risulta primo della lista di priorità a cui appartiene (di una data CPU). Questo tipo di processo viene rimpiazzato se:

- subisce prelazione da un altro processo con maggiore priorità real-time;
- Esegue un'operazione bloccante ed è messo in sleep;
- il processo è stato fermato o ucciso;
- Il processo rilascia volontariamente la CPU lanciando la chiamata di sistema `sched_yield()`;
- Il processo è in Round Robin real-time, e ultimato il suo quanto di tempo viene prelazionato.

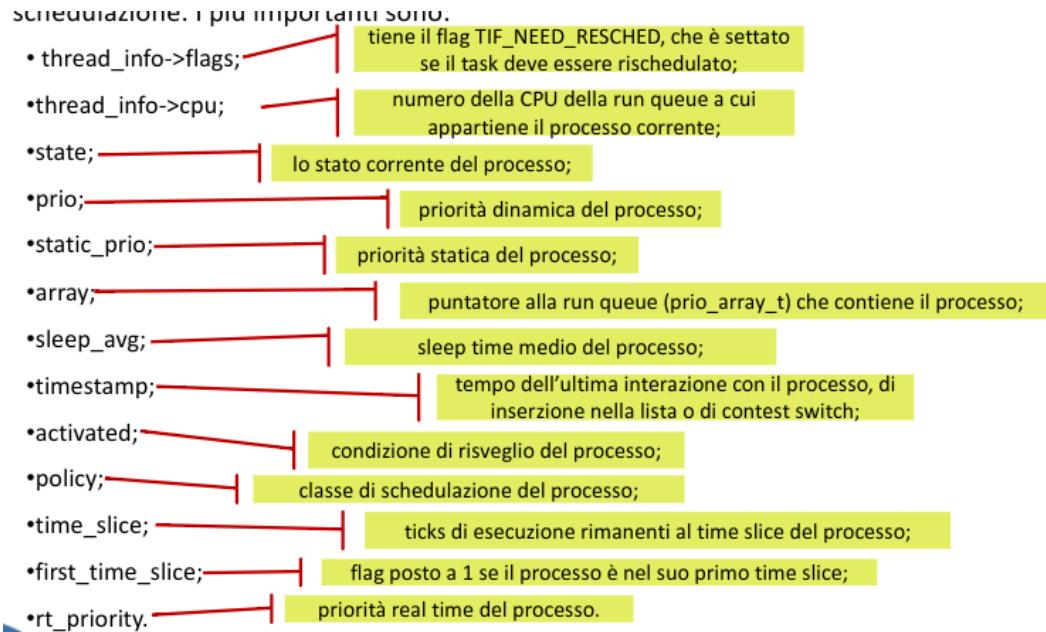
7.3 Strutture dati usati dallo scheduler

La struttura a run queue è la più importante fra quelle utilizzate dallo scheduler su Linux. Ogni CPU presente nel sistema ha la propria run queue.

La macro `this_rq()` tiene l'indirizzo della run queue della CPU locale, mentre la macro `cpu_rq(n)` fornisce l'indirizzo della run queue della CPU con indirizzo n.

7.3.1 Principali campi della struttura

I principali campi della struttura sono:	
• <code>unsigned long nr_running;</code>	tiene il conto dei processi running presenti nella ruqueue;
• <code>struct mm_struct *prev_mm;</code>	usato durante uno switch per indirizzare il descrittore di memoria del processo che sta per essere rimpiazzato;
• <code>struct task_struct *curr;</code>	puntatore al descrittore del processo che sta eseguendo;
• <code>struct task_struct *idle;</code>	puntatore al descrittore del processo swapper per la CPU corrente;
• <code>struct prio_array *active, *expired;</code>	puntatori alle liste di processi attivi e a quella dei processi "scaduti";
• <code>struct prio_array arrays[2];</code>	array contenente i set di processi attivi e scaduti;
• <code>struct sched_domain *sd;</code>	punta al dominio di schedulazione della CPU corrente;
• <code>struct task_struct *migration_thread;</code>	puntatore al descrittore del kernel thread "migration";
• <code>struct list_head migration_queue;</code>	lista dei processi che devono essere rimossi dalla run queue list.



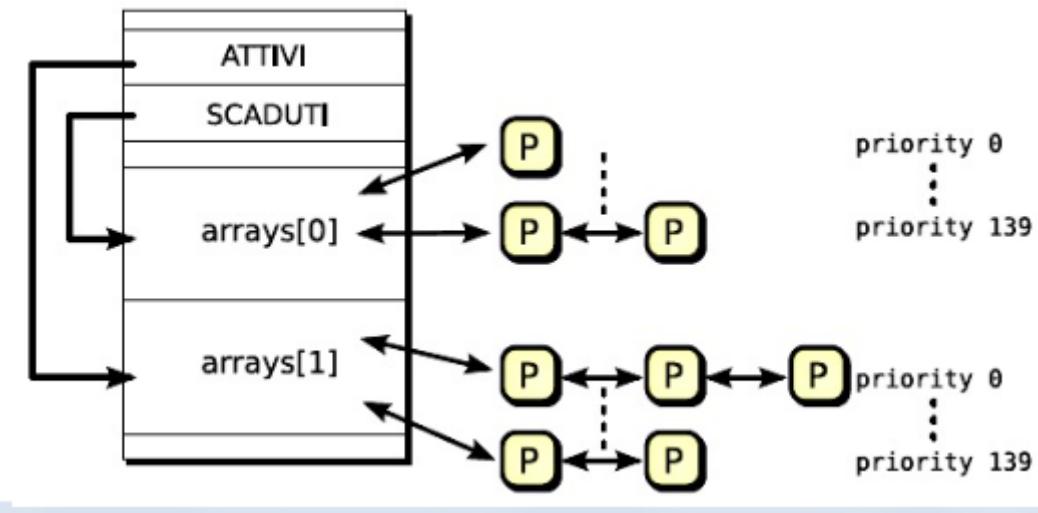
7.4 Scheduling dei processi nel dettaglio

Il campo più importante della struttura runqueue è quello legato alla lista di processi eseguibili (run queue).

Ogni processo runnable appartiene ad una e una sola run queue → finchè il processo rimarrà in quella lista, solo la CPU corrispondente a quella run queue potrà eseguirlo.

La run queue prevede inoltre due puntatori a delle strutture di tipo `prio_array` che detengono la lista dei processi attivi e scaduti.

Esse includono 140 nodi iniziali (heads) di doubly linked list (una per priorità), una mappa di priorità (priority bitmap) ed un contatore che conta i processi della struttura. Periodicamente i due puntatori vengono scambiati tra di loro.



7.5 Funzioni utilizzate dallo scheduler

Lo scheduler utilizza 5 funzioni fondamentali nella gestione dei processi:

- `scheduler_tick()` → utilizzata per aggiornare il `time_slice` del processo current;
- `try_to_wake_up()` → risveglia un processo dormiente;
- `recalc_task_prio()` → aggiorna la priorità del processo;
- `schedule()` → seleziona il processo che deve eseguire;
- `load_balance()` → tiene le run queue dei vari processi bilanciate

7.6 Aggiornamento del time slice di processi Real-time

Per i processi real-time le istruzioni di aggiornamento si distinguono a seconda che il processo current sia:

- FIFO
 - lo scheduler non fa nulla (il processo non può subire preemption da un processo con priorità uguale o minore)
 - Non avrebbe senso tenere il suo contatore aggiornato
- RR
 - Se il quanto è scaduto dopo averlo decrementato
 - * viene ricalcolato il time slice del processo;
 - * viene azzerata la condizione di esecuzione nel primo time slice;
 - * viene posato il flag `TIF_NEED_RESCHED` ad 1.

Prima di uscire, la funzione pone `current` in coda ai processi dello stesso anello, rilascia il lock e forza l'invocazione dello scheduler quando l'esecuzione rientra dall'interrupt del timer

7.7 Rendere un processo running

I processi possono attendere la realizzazione di un certo evento, e dunque si sospendono in uno stato `TASK_INTERRUPTIBLE` o in uno stato di `TASK_UNINTERRUPTIBLE`.

Qualora un processo in uno di questi stati volesse risvegliarsi, la routine del kernel adibita a questa funzione dovrà utilizzare il metodo `try_to_wake_up`.

Il metodo riceve i seguenti parametri:

- il descrittore del processo da risvegliare;
- una maschera dello stato del processo che può essere risvegliato;
- un flag che proibisce al processo risvegliato di prelazionare il processo correntemente running

7.7.1 Funzione `schedule()`

Ha il compito di scegliere il processo più adatto all'esecuzione. Due tipo di invocazione:

- *modalità diretta* → si presenta quando un task deve sospendersi per mancanza di una risorsa;
- *modalità lazy* → attivando il flag `TIF_NEED_RESCHED` si forza l'invocazione dello scheduler al successivo ritorno in User Mode.

Modalità diretta

In questo caso la routine del kernel (a seguito di una chiamata di sistema) deve:

- Inserire current nella coda di attesa della risorsa associata che gli spetta;
- Cambiare lo stato di current in `TASK_INTERRUPTIBLE` o

`TASK_UNINTERRUPTIBLE`, a seconda del processo;

- Invocare la funzione `schedule()`;
- Controllare periodicamente se la risorsa è disponibile;
- Quando la risorsa è disponibile, risvegliare il processo rimuovendolo dalla coda di attesa.

È il caso di alcuni driver che eseguono lunghe iterazioni. Questi task controllano ad ogni iterazione se il valore del flag di current è impostato su `TIF_NEED_RESCHED` , nel qual caso chiamano la funzione `schedule()` per rilasciare volontariamente l'uso della CPU.

Modalità lazy

Ci possono essere vari casi in cui si effettua una chiamata della funzione in questo modo:

- current ha ultimato il suo quanto temporale, `scheduler_tick()`

quindi setta il flag `TIF_NEED_RESCHED`;

- Un task, che è stato risvegliato dalla routine del kernel, ha

priorità maggiore del processo current, in questo caso l'azione è compiuta dalla funzione `try_to_wake_up()`;

- Quando è invocata la chiamata di sistema

`sched_setscheduler()`.

7.7.2 Schedule() prima e dopo il context switch

La funzione chiama prev e next rispettivamente il task che deve essere sostituito e quello che eseguirà successivamente. Essa esegue alcune operazioni fondamentali prima del context switch:

- Calcola il tempo che il processo sottrae al proprio time slice;
- Verifica se il processo prev sia un processo che sta per essere terminato;
- Verifica l'esistenza di processi runnable
 - Se non ce ne sono, next viene settato sullo swapper e attivi e scaduti vengono scambiati;
 - Se ce ne sono, verifica se appartiene agli attivi oppure agli scaduti.
- Determina qual è il processo della run queue che effettivamente ha priorità maggiore;
- Associa alla variabile locale next il puntatore al descrittore del processo che sostituirà prev;
- Invoca la funzione `context_switch()` che esegue lo scambio di contesto

tra i due processi.

La funzione `finish_task_switch(prev)` controlla se il processo passato come parametro è uno zombie, in caso positivo avvia le procedure per eliminarlo. Le ultime istruzioni controllano se altri processi hanno posto il flag di quello corrente a `TIF_NEED_RESCHED`; in tal caso la funzione viene eseguita da capo, altrimenti termina

7.8 Gestione dei Page Frame

Linux opta per Page Frame di dimensioni di 4Kb, per 2 motivi:

- I Page Fault vengono gestiti in modo più efficiente;
- Alcune operazioni risultano più efficienti quando si utilizza un page frame di piccole dimensioni

7.9 Page Descriptor

Il kernel tiene traccia delle informazioni relative ad un page frame, incapsulandole all'interno di un **page descriptor**. Ciascun descrittore ha una dimensione di 32 byte ed è contenuto all'interno di un array chiamato **mem_map**.

Data la dimensione limitata, lo spazio occupato dall'insieme dei descrittori occupa meno dell'1% della memoria disponibile del sistema.

Dato un indirizzo lineare **addr** è possibile recuperare il page descriptor che gli corrisponde attraverso la macro **virt_to_page(addr)**

Dato l'indirizzo di un page frame **pfn** è possibile recuperare l'indirizzo del suo page descriptor mediante la macro **pfn_to_page(pfn)**

7.9.1 Campi di un Page Descriptor

Rivesto una certa importanza:

- **_count** → contatore degli utilizzi del page frame
 - Impostato a -1 indica che il page frame è libero;
 - Con valore ≥ 0 indica il numero di processi che lo stanno utilizzando;
- **flags** → descrive lo stato del page frame.

7.10 NUMA

In alcune architetture non tutte le CPU accedono a tutte le zone di memoria con la stessa velocità.

La memoria è divisa in nodi ed il kernel alloca per ogni CPU, quei nodi ai quali essa può accedere con maggiore velocità, riducendo gli accessi ai nodi più lenti.

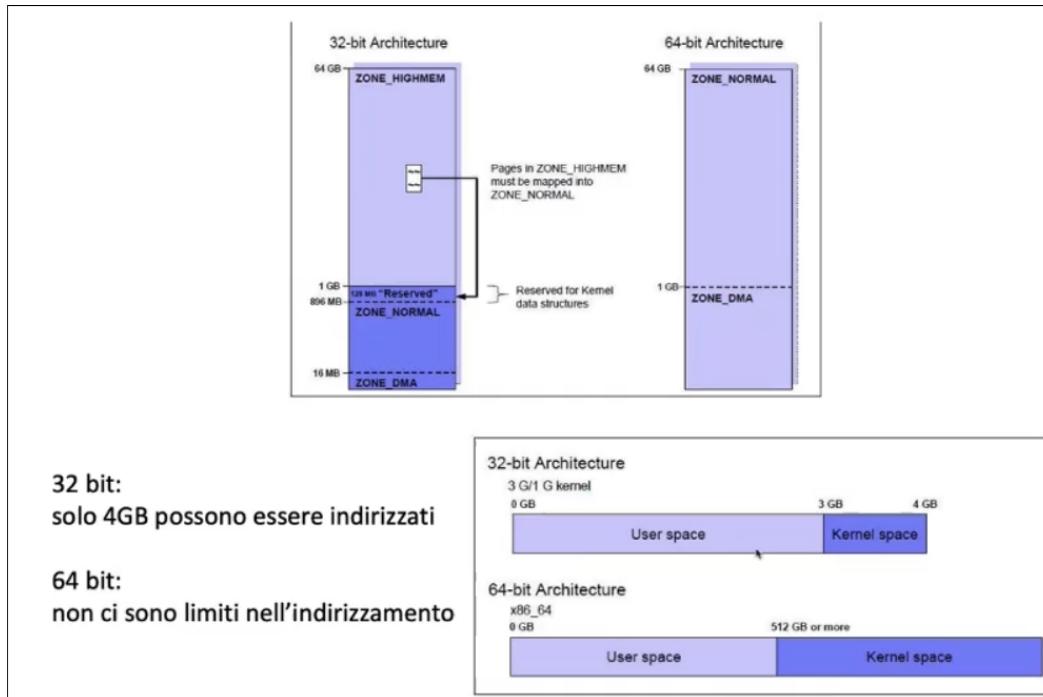
La memoria contenuta all'interno di ciascun nodo è poi suddivisa in zone.

Linux considera la memoria come contenuta in un unico modo → questo gli permette di mantenere la compatibilità con i sistemi che utilizzano uno schema NUMA (Non Uniform Memory Access)

7.11 Zone di memoria

Linux gestisce la memoria in 3 diverse regioni:

- ZONE_DMA;
- ZONE_NORMAL;
- ZONE_HIGHMEM;



Linux adotta 2 diversi metodi per rispondere alle richieste di allocazione di memoria:

- Se c'è memoria sufficiente le pagine richieste vengono allocate direttamente;
- Se le pagine non possono essere allocate a causa di scarsa disponibilità di memoria, viene attivata una procedura di *page reclaiming*.

Il secondo approccio non è fattibile per quei KCP (kernel control path) che non possono eseguire operazioni bloccanti. In questo caso inviano una atomic request, e nel caso in cui questa fallisca il KCP termina con un errore. Al fine di limitare questi casi il kernel si riserva un numero limitato di pagine per gestire le atomic request.

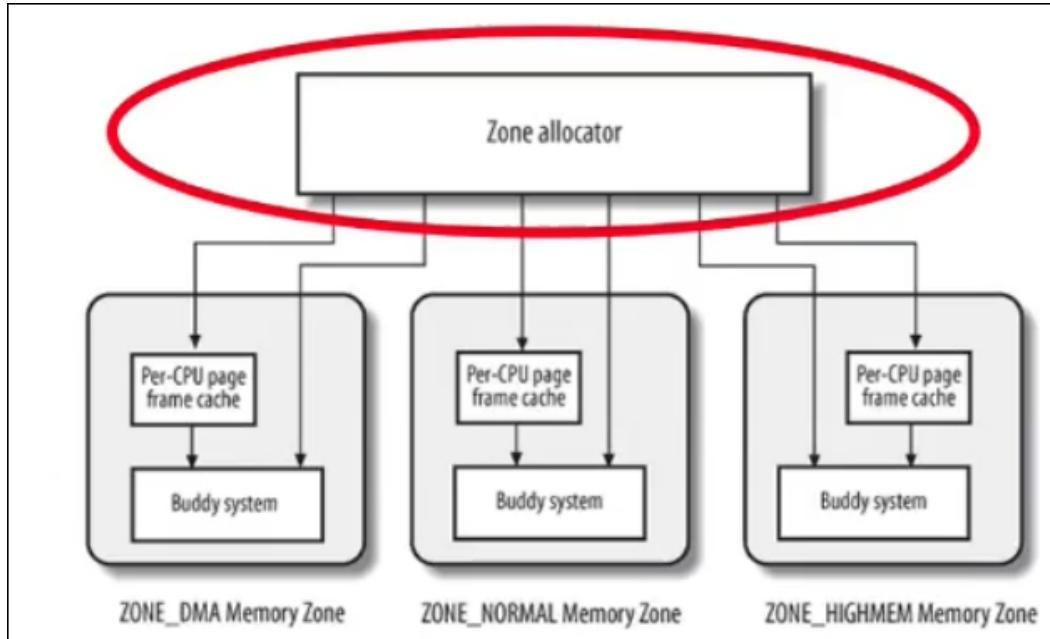
$$\text{reserved pool size} = [\sqrt{16 \times \text{directly mapped memory}}]$$

7.12 Zoned Page Frame Allocator

Il sottosistema del kernel che gestisce le richieste di allocazione di gruppi di page frame contigui, prende il nome di **zoned page frame allocator**.

All'interno di ciascuna zona, i page frame vengono gestiti da un componente che prende il nome di Buddy System.

Un limitato numero di pagine viene riservato in alcune cache, che prendono il nome di *Per-CPU page frame cache*



Lo Zone allocator riceve le richieste di allocazione e deallocazione di memoria. Per allocazione lo ZA ricerca una zona di memoria che comprende un numero di pagine necessario per soddisfare la richiesta. All'interno di ogni zona i page frame sono gestiti dal Buddy System.

Lo ZA è il frontend del *kernel page frame allocator*. Le richieste di allocazione vengono sempre ricondotte a questo componente, che ha il compito di vigilare su diversi aspetti:

- I page frame allocati non possono essere presi dal pool di pagine riservate dal kernel per le allocazioni atomiche;
- Quando non c'è disponibilità di pagine, invoca la procedura di page reclaiming;
- Ottimizza i page frame allocati nella ZONE_DMA, viste le sue dimensioni limitate.

Il kernel deve quindi stabilire una robusta strategia ed efficiente per allocare gruppi di page frame contigui. Per fare ciò, deve affrontare il problema della *frammentazione esterna*, che viene risolto dal Buddy system

7.12.1 Per-CPU page frame cache

Il kernel spesso rilascia e richiede singole pagine. Per migliorare le prestazioni, ogni zona di memoria definisce una *per-CPU page frame cache*, la quale include alcune pagine di memoria preallocate per essere usate per singole richieste di memoria da parte della CPU locale.

7.13 Kernel Mapping di page frame

La memoria indirizzata direttamente dal Kernel, si limita ai primi 896 Mb. I page frame oltre gli 896 Mb vengono gestiti in modo indiretto.

La allocazione di questi page frame è gestita dalle funzioni `alloc_page()` e `alloc_pages()`. Esse non restituiscono l'indirizzo lineare del page frame, ma l'indirizzo lineare del corrispondente page descriptor, il quale è sempre conservato nella memoria bassa.

Gli ultimi 128 Mb nello spazio degli indirizzi lineari del kernel è riservato alla gestione di questi page frame, la quale viene effettuata con 3 diversi meccanismi:

- Permanent Kernel Mapping;
- Temporary Kernel Mapping;
- Non Contiguous Memory Allocation;

7.14 Il problema della frammentazione esterna

Linux adotta un sistema dedicato alla gestione dei page frame al fine di risolvere il problema della frammentazione esterna. Si possono adottare 2 filosofie diverse per far fronte a questo aspetto:

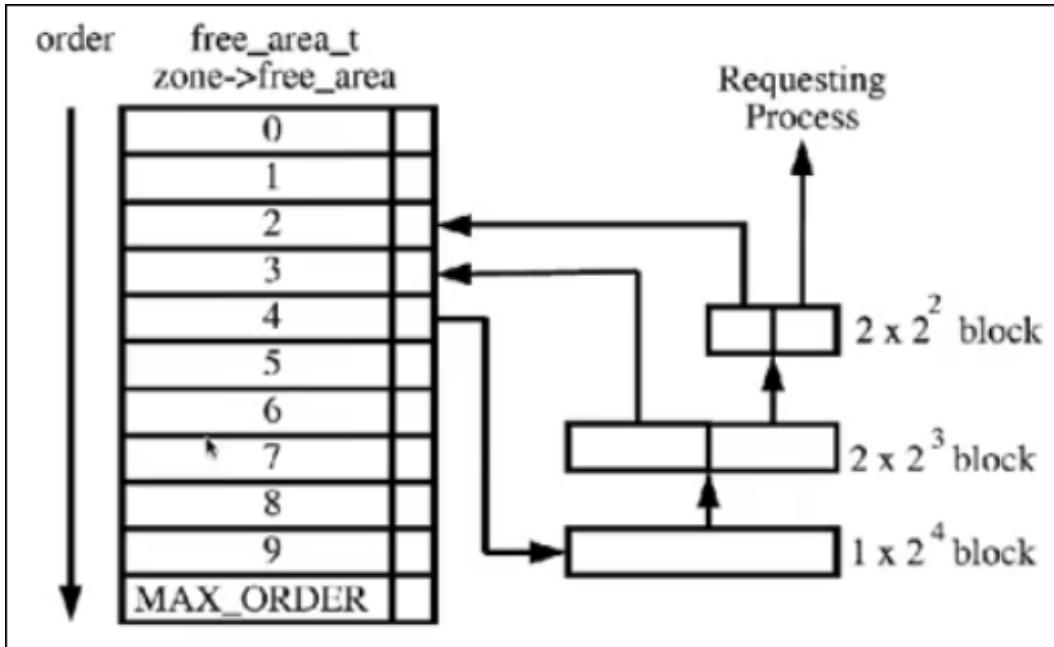
- Mappare intervalli di indirizzi lineari su zone di memoria non contigue;
- Implementare un componente dedicato, che tenga traccia delle regioni contigue disponibili in memoria centrale.

Linux opta per la seconda strategia, proprio per alcuni vantaggi che questa offre:

- La memoria riserva in alcuni casi deve essere continua;
- Riduce gli aggiornamenti delle Page Table;
- Permette la gestione di grandi aree di memoria con pagine di 4Mb

7.15 Il Buddy System

I page frame in memoria sono organizzati in 11 liste $k = 0, 1, \dots, 10$, che contengono gruppi di 2^k pagine. Dato un gruppo di ordine k , l'indirizzo fisico del primo page frame di tale gruppo è un multiplo della dimensione della pagina (4 kb = 4096 byte = 2^{12} byte)



7.15.1 Filosofia del Buddy System

Quando un processo necessita di un dato numero di page frame, fissa un ordine k e sottopone al buddy system una richiesta.

Se esso è in grado di evaderla, restituisce un indirizzo fisico dei page frame allocati, altrimenti scorre le liste degli ordini superiori, e divide un gruppi più grande in gruppi più piccoli.

In caso contrario, incrementa k di 1 ed effettua nuovamente il controllo. Se è disponibile un gruppo nella lista di ordine $k+1$:

- Dimezza il gruppo in 2 sottogruppi di ordine k ;
- Alloca un gruppo, e inserisce l'altro nella lista di ordine k .

In caso contrario incrementa nuovamente k ;

Supponiamo trovi un gruppo di ordine $k+a$, con $a > 1$:

- Partiziona il gruppo in due sottoruppi di dimensione $k+a-1$;
- Inserisce il primo sottogruppo nella lista di ordine $k+a-1$;
- Suddivide ulteriormente il secondo sottogruppo in due di dimensioni $k+a-2$;
- Dopo a iterazioni di questo processo, arrivato all'ordine k , alloca un gruppo e accoda l'altro.

7.15.2 Strutture dati

Linux considera la memoria del sistema divisa in 3 zone, pertanto utilizza un diverso buddy system per ciascuna di esse.

Ciascuno dei 3 buddy system fa riferimento a delle strutture dati:

- array `mem_map` → ciascuna delle 3 zone di memoria è rappresentata da una porzione di questo array. Ciascuna zona è caratterizzata da uno zone descriptor che ne indica il primo elemento ed il numero di elementi;
- un array di tipo `free_area`, conservato nel campo `free_area` dello zone descriptor.
 - Questo array ha 11 posizioni, una per ciascun ordine.

Il k -esimo elemento del secondo array rappresenta la testa di una lista doppiamente linkata che punta al primo page frame del primo gruppo di page frame di ordine k

7.15.3 Liberazione di un blocco

La funzione incaricata di liberare i blocchi per un buddy system è `_free_page_bulk()`.

Essa opera con gli interrupt disabilitati e dopo che un opportuno spin lock è stato attivato. Dopo aver liberato un blocco la funzione controlla se è possibile fondere il gruppo di pae frame appena liberato con il suo gemello, verificando 3 condizioni:

- Entrambi i blocchi devono avere la stessa dimensione b (lo stesso ordine k);
- Gli indirizzi dei due blocchi si riferiscono a regioni di memoria contigue;
- L'indirizzo fisico del primo page frame del primo blocco è un multiplo di $2 \times b \times 2^{12}$

8 Lezione del 29-03

8.1 Per-CPU Page Frame Cache

Molto spesso il kernel richiede una singola pagina, e far riferimento al buddy system, per allocazioni di questa taglia crea un inutile overhead. Il kernel quindi riserva un pool di page frame come cache.

Il kernel considera 2 tipi di cache di page frame:

- **hot cache** → pagine che vengono usate da un processo subito dopo essere state richieste;
- **cold cache** → possono essere utilizzate anche non immediatamente dopo l'allocazioni (utilizzate spesso dalle DMA);

La struttura dati che le gestisce è un array di strutture dati di tipo `per_cpu_pageset`.

Anche questo array è contenuto nel memory zone descriptor

8.2 La funzione `_alloc_pages()`

Riserva gruppi di page frames in una specifica zona di memoria

Signature = `_alloc_pages(gfp, order, zonelist)`

Esso si serve inoltre di una funzione `zone_watermark_ok()`, per calcolare una soglia minima (*mark*), in base alla quale stabilire se i page frame richiesti possono essere allocati o meno; Restituisce 1 se il valore è maggiore della soglia minima

Quando la funzione `alloc_pages()`, si effettuano una serie di step prima di restituire un indizzo valido o NULL, a seconda della disponibilità di memoria nel sistema:

- Effettua un primo scan della zona di memoria;
- Se la funzione non termina durante il primo scan, non c'è abbastanza memoria;
 - Quindi sveglia i kernel thread kswapd per effettuare un page reclaiming;
- Abbassa la soglia mark ed effettua un secondo scan;
- Se nemmeno in questo caso la funzione termina prima della fine dello scan, la memoria disponibile è insufficiente e se il KCP è autorizzato fa ricorso alle pagine della `low-on-memory reserver`, terminando con un NULL (in caso di insuccesso);
- Se i passi precedenti hanno liberato delle pagine, viene effettuato un terzo scan;
- Se lo scan fallisce e il processo può essere messo in sleep, lo sospende
 - Se il processo non può essere sospeso, cerca un processo da uccidere, con la funzione `out_of_memory()`.

8.3 La funzione `_free_pages()`

Riceve in input:

- il page descriptor del primo page frame del gruppo;
- L'ordine del gruppo di page frame.

La funzione:

- verifica che il page descriptor faccia riferimento ad una regione della memoria dinamica;

- Decrementa il contatore page → `_count` (Se questo è maggiore di 0 termina);
- Se l'ordine k è 0, si tratta di un page frame della hot cache, per cui chiama la funzione `free_hot_page()`;
- Se l'ordine k è maggiore di 0, rilascia il gruppo al buddy system di competenza della zona di memoria cui il gruppo appartiene.

8.4 Memory Area Management

Lo spazio all'interno delle pagine potrebbe essere parzialmente inutilizzato dando origine ad una frammentazione interna

Alcune versioni di Linux precedenti la 2.6, contenevano tale fenomeno permettendo allocazioni di blocchi la cui dimensione fosse potenza di 2 (distribuzione geometrica). In questo modo la frammentazione interna è sempre minore del 50%.

Linux 2.6 adotta un sistema simile a quello implementato dal sistema Solaris2.4, implementando uno Slab Allocator

8.5 Lo Slab Allocator nel dettaglio

Lo Slab Allocator tratta le aree di memoria come degli oggetti che vengono allocati e deallocati all'interno di contenitori detti slab (lastre).

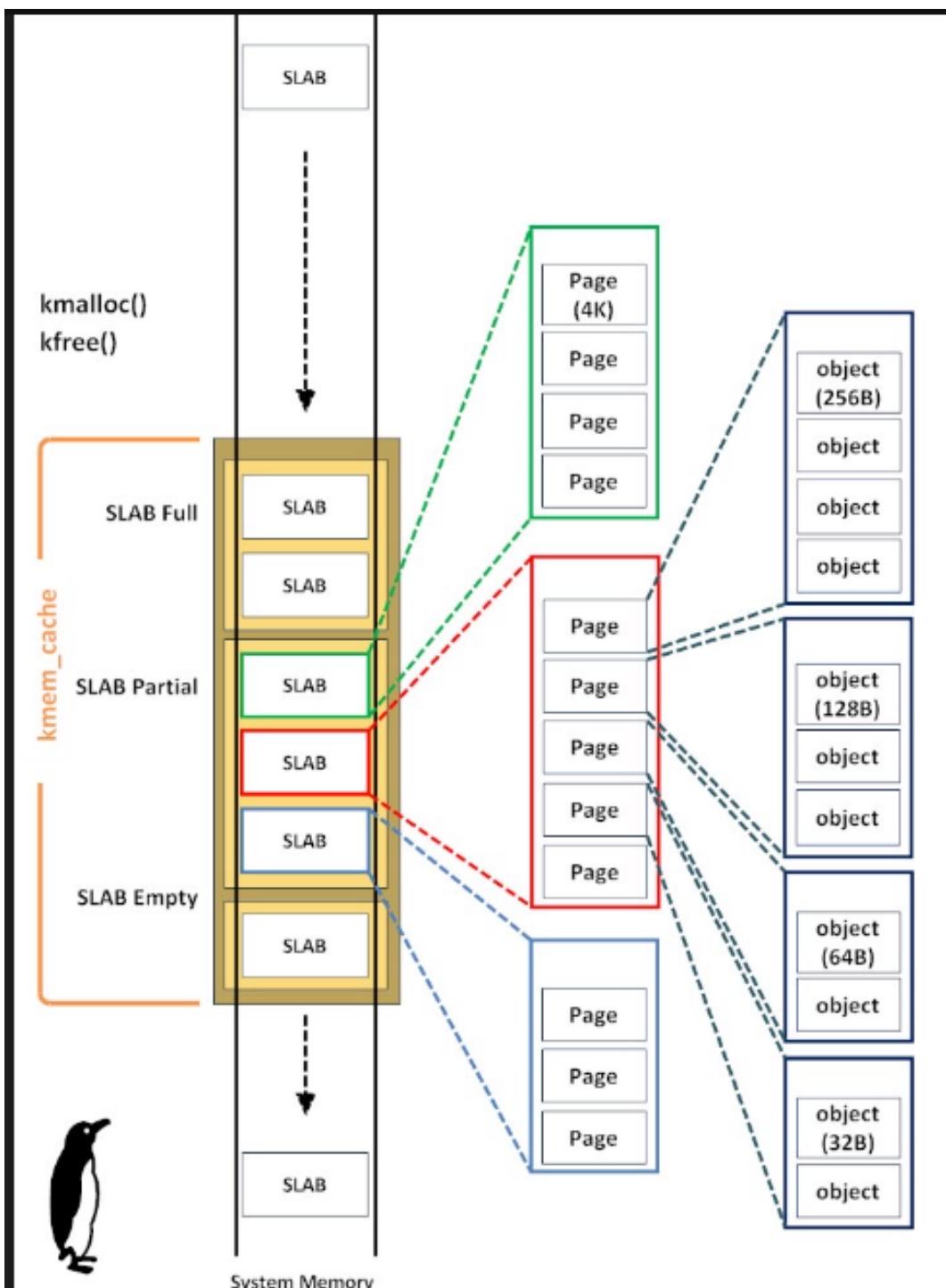
Esso nasce da alcune considerazioni fondamentali:

- Sulla base delle strutture dati da memorizzare in una zona di memoria è possibile ottimizzare il processo di allocazione e deallocazione
 - Le aree di memoria sono viste come oggetti, dotate di un costruttore e di un distruttore.
 - Gli oggetti deallocati non sono immediatamente distrutti
- Le funzioni del kernel richiedono con elevata frequenza l'allocazione di alcune strutture dati (es. creazione di un processo). Oggetti allocati e deallocati possono essere riutilizzati
- Le allocazioni possono essere gestite in base alla frequenza. Quelle molto frequenti con l'allocazione di oggetti, quelle meno frequenti con la distribuzione geometrica delle dimensioni delle aree di memoria.
- Gli indirizzi delle zone di memoria non si concentrano necessariamente intorno a multipli di potenze di 2;
- Il buddy system (hot cache) induce una alterazione dei contenuti delle cache HW (footprint).

Lo Slab Allocator, gestisce delle regioni di memoria, dette Cache. All'interno di ciascuna di queste sono allocate un certo numero di lastre. A loro volta ciascuna di queste contiene un certo numero di oggetti.

Le Cache e le Slab sono caratterizzate rispettivamente da un Cache Descriptor e da uno Slab Descriptor.

Periodicamente il kernel fa uno scan delle cache e rilascia i page frames che corrispondono a lastre vuote

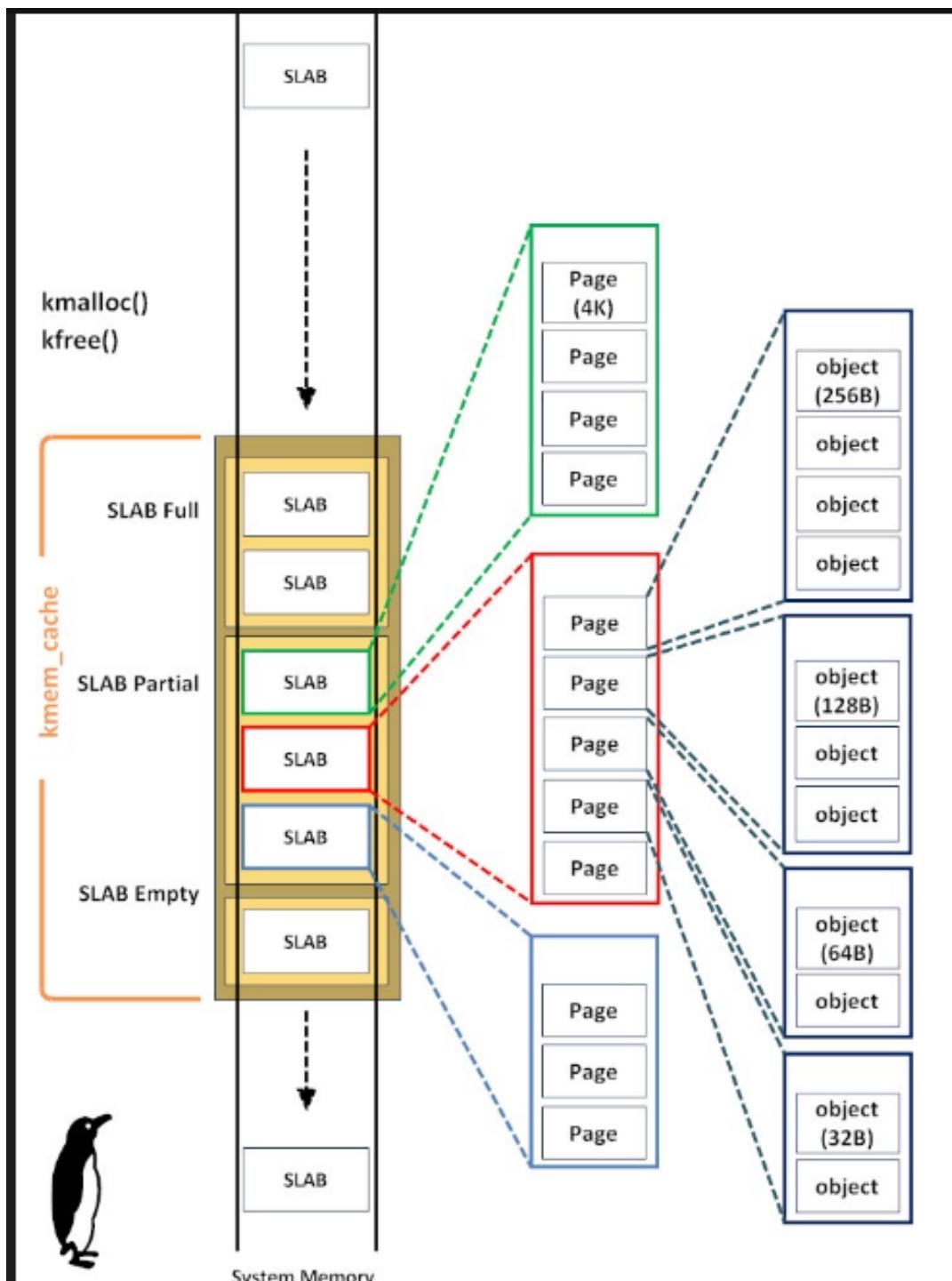


8.6 Cache Generiche e Specifiche

Divise in 2 categorie:

- Generiche → utilizzate solo dallo Slab Allocator, per i propri scopi

- ad es per contenere i cache descriptor (`kmem_cache`);
- Specifiche → contengono slab con oggetti tutti dello stesso tipo
 - ad es l'oggetto per i file aperti, o per le page table dei processi



8.7 Allocazione e Deallocazione di Slab

Quando lo Slab allocator deve allocare una nuova slab esegue in cascata una serie di step:

- Richiede allo *Zoned Page Frame Allocator* un blocco di page frame da associare alla slab, specificando quale cache necessita della memoria;
- Associa la nuova slab alla cache, se e solo se
 - È stata inoltrata una richiesta di allocazione per un nuovo oggetto;
 - Nella cache non è già presente un oggetto libero.

Poichè il kernel tiene traccia di come viene utilizzata la memoria, deve:

- Poter determinare efficientemente se dei page frame sono utilizzati dallo slab allocator;
- Ottenere in modo efficiente il corrispondente cache descriptor e slab descriptor.

Le informazioni necessarie sono inserite nel campo **Lru** del page descriptor

Lo Slab Allocator libera una slab in 2 casi:

- Al suo interno ci sono troppi oggetti liberi;
- La slab è completamente inutilizzata (viene effettuato un check periodico)

Se gli oggetti di una slab sono forniti di un metodo distruttore, questo viene invocato per tutti gli oggetti al suo interno e i page frame corrispondenti alla slab vengono liberati.

8.8 Gli Object Descriptor

Ciascun oggetto contenuto in una slab è dotato di un **Object Descriptor**.

Un Object Descriptor è un valore intero. Ha senso se l'oggetto è libero, in quanto indica la posizione del successivo oggetto libero, implementando una sorta di lista linkata. Come gli altri descrittori può trovarsi:

- Fuori dalla slab, ossia nella cache generica puntata dal cache descriptor in cui l'oggetto si trova;
- Dentro la slab, immediatamente prima dell'oggetto che descrive

8.9 Slab Coloring

Gli oggetti gestiti dallo slab allocator sono allineati in memoria, ossia il loro indirizzo è multiplo di una potenza di 2 (*aln*). Solitamente si sceglie la lunghezza di una parola macchina, quindi 4 in un sistema a 32 bit.

Quando gli oggetti vengono associati a linee di cache HW finiscono, con molta probabilità, sulle stesse linee di cache. A ciascuna slab è associato un colore *col* e tutti i suoi oggetti vengono allineati a partire dall'indirizzo *col* \times *aln* + *dsize*, dove *dsize* rappresenta la somma delle dimensioni della slab e di tutti i suoi oggetti.

8.10 Local Cache e Memory Pool

Il kernel riserva in anticipo alcune risorse, tra cui:

- Local Cache → cache locale di slab per ciascuna CPU, il sistema ricorre allo slab allocator solo quando non riesce a soddisfare le richieste sulla base di queste risorse.
- Memory Pool → insieme di page frame riservati a priori per uno specifico componente del kernel
 - Vengono utilizzate solo quando il sistema è sovraccarico e la memoria è insufficiente.

Si tratta di un insieme diverso dal reserved pool, che può essere utilizzato solo per soddisfare le atomic memory allocation request

8.11 Processi utente e gestione della memoria

Il Kernel gestisce la memoria dinamica facendo ricorso a diverse funzioni.

Le strategie adottate dal kernel, garantiscono una gestione efficiente delle risorse di memoria, per 2 ragioni principali:

- Il Kernel ha sempre la priorità massima;
- Gode della massima fiducia (si suppone sia bug free e non implementi azioni malevoli).

La gestione della memoria per conto dei processi funziona in maniera diversa:

- Le richieste di memoria dei processi utente sono considerati non urgenti (ritardati il più possibile);
- Il kernel deve verificare il corretto comportamento dei processi utente.

8.12 Lo spazio degli indirizzi

Lo spazio degli indirizzi di un processo è l'insieme di tutti gli indirizzi lineari che il processo può utilizzare.

Ciascun processo può gestire un limitato sottoinsieme di tutti gli indirizzi lineari disponibili.

Il kernel può aggiungere o eliminare dinamicamente un intervallo di indirizzi lineari, per diverse ragioni:

- L'esecuzione di un comando da shell, genera un nuovo processo;

- Un processo invoca una *exec()* per eseguire un programma differente;
- Un processo crea una ragione di memoria condivisa;
- Un processo chiama una delle funzioni *malloc()*, *calloc()*, *free()*.

8.13 Regioni di memoria

Gli intervalli di indirizzi lineari sono rappresentati dal kernel mediante regioni di memoria. Ciascuna regione di memoria è caratterizzata da:

- Il primo indirizzo lineare in essa contenuto (multiplo di 4096);
- La sua lunghezza (multiplo di 4096);
- Dei diritti di accesso.

Per il kernel è fondamentale conoscere il proprietario di ciascuna regione di memoria al fine di poter gestire correttamente i page fault:

- Alcuni sono causati da errori del programma;
- Altri sono causati dal fatto che una pagina richiesta non sia in memoria.

Ciascuna regione di memoria è caratterizzata da un memory descriptor, il cui riferimento *mm* è nel process descriptor del processo, che ne risulta proprietario.

Le regioni di memoria, che appartengono ad uno stesso processo non si possono sovrapporre.

Quando una nuova regione di memoria viene allocata o deallocata, in modo adiacente ad una già esistente per lo stesso processo, il kernel verifica:

1. Se è possibile allargare una regione esistente;
2. Se è necessario creare una nuova regione;
3. Se è possibile ridimensionare le regioni di memoria rimanenti;
4. Se è necessario dividere la regione in due sotto-regioni.

8.14 Strutture dati

Il memory descriptor possiede un campo *map_count* che indica il numero di regioni appartenenti al processo ed un puntatore *mmap* ad una lista concatenata

Tutte le regioni appartenenti allo stesso processo sono organizzate in una lista concatenata.

Le regioni nella lista compaiono in ordine crescente rispetto all'indirizzo lineare di inizio. Fra una regione e l'altra possono esserci regioni allocate a processi differenti

8.15 Ricerca di una regione di memoria

Una delle operazioni più frequenti per il kernel è la ricerca della regione che contiene un determinato indirizzo.

Una delle strategie adottate dal kernel per la ricerca delle regioni è la ricerca lineare scorrendo la lista. Il kernel comincia dalla testa e arresta la ricerca appena trova una regione che termina dopo l'indirizzo cercato.

Questo approccio è efficiente solo quando un processo ha un numero limitato di regioni di memoria.

Linux 2.6 adotta anche un approccio basato su alberi **red-black**, la cui radice è puntata dal campo `mm_rb` del memory descriptor

8.15.1 Alberi red-black

Alberi binari di ricerca estesi. La caratterizzazione degli alberi red-black avviene attraverso la formulazione di 4 proprietà vincolanti.

1. Ogni nodo dell'albero è *rosso* o *nero*;
2. OGni foglia dell'albero (NIL) è nera;
3. Se un nodo è rosso entrambi i suoi figli sono neri;
4. Dato un nodo, tutti i percorsi discendenti che raggiungono una foglia contengono lo stesso numero di nodi neri

La ricerca di una regione attraverso l'albero ha sempre un costo pari a $O(\log(n+1))$ dove n è il numero di regioni. Le operazioni di aggiornamento dell'albero hanno costo logaritmico.

Aggiungere un nuovo livello nell'albero significa aumentare di una iterazione il costo dell'algoritmo di ricerca.

8.16 Linux e la ricerca di una regione di memoria

Adotta entrambe le strutture dati per gestire le regioni di memoria.

La lista e l'albero utilizzano puntatori distinti, che puntano alle stesse regioni di memoria.

Generalmente, la lista è utilizzata per scorrere tutto lo spazio degli indirizzi, mentre l'albero red-black è usato per cercare una regione che contenga un determinato indirizzo.

8.17 Diritti di accesso ad una regione di memoria

Una regione di memoria consiste in un insieme di pagine contigue.

Ciascuna pagina è accompagnata da *flag* che ne caratterizzano l'utilizzo:

- Read, Write, Present, User, Supervisor nella Page Table entry;
- Locked, Dirty, LRU, SLAB nel page descriptor.

A questi si aggiunge un'ulteriore serie di flag espressa dal campo `vm_flags` nel memory region object (di tipo `vm_area_struct`)

8.18 Demand Paging

Tecnica di allocazione dinamica della memoria che tenta di ritardare quanto più possibile la reale assegnazione delle pagine fisiche.

Questo ha senso, poichè un processo non utilizza mai tutte le pagine richieste sin dall'inizio della sua esecuzione; inoltre, ci sono pagine che non vengono mai utilizzate.

Questa tecnica offre 2 principali vantaggi:

- Mantenere un più alto numero di pagine disponibili;
- Aumentare il throughput del sistema, svolgendo lo stesso lavoro con un minor numero di pagine di memoria.

Il principale svantaggio è che i page fault vanno gestiti e producono overhead. Tuttavia, per il principio di località, i processi tendono ad utilizzare sempre un ristretto gruppo di pagine, rendendo i page fault abbastanza rari.

8.19 Page Reclaiming

Linux limita al massimo i controlli sulla quantità di memoria che può essere allocata ad un processo o ad un kernel thread, così da permetterne un uso quanto più possibile efficiente.

Il modo, in cui la RAM viene gestita dipende dal carico del sistema:

- *Basso carico* → molte pagine sono utilizzate per tenere in memoria informazioni caricate dal disco;
- *Elevato carico* → le pagine di memoria vengono principalmente utilizzate per le cache dei processi e del kernel

Le pagine allocate ad una cache in memoria, non vengono gestite in modo diretto, poichè non è dato sapere quando un processo non utilizzerà più i dati in tale cache.

Linux implementa un meccanismo di [page reclaiming](#), che gli permette di recuperare le pagine, quando se ne crea la necessità.

8.20 Selezione di una pagina target

Linux seleziona le pagine da liberare attraverso un [Page Frame Reclaiming Algorithm](#) (PFRA).

Le pagine che rappresentano un possibile candidato per il PFRA devono essere pagine non libere, ossia non devono appartenere già al pool di pagine disponibili per il Buddy System.

Al fine di selezionare una eventuale pagina target, il PFRA distingue le pagine allocate in 4 diverse categorie:

- Non reclamabili → pagine che non possono essere liberate per diverse ragioni
 - Pagine libere;
 - Pagine riservate;
 - Pagine allocate direttamente dal kernel;
 - Pagine locked;
- Swappabili → pagine il cui contenuto deve essere scritto sul disco prima che esse vengano liberate;
- Sincronizzabili → pagine che fanno da cache per dati su disco o che rappresentano un buffer per qualche device. Esse devono essere allineate con il contenuto di cui sono cache, prima di essere liberate;
- Scartabili → pagine non utilizzate, come quelle allocate per oggetti vuoti nelle lastre dello slab allocator.

Un aspetto importante, di cui il PFRA deve tenere conto quando seleziona una pagina, è se essa è condivisa o meno

8.20.1 Principi alla base di un PFRA

La reale difficoltà consiste nel progettare un algoritmo che sia in grado di funzionare bene anche in contesti molto diversi fra loro:

- Sistemi desktop → le richieste di memoria sono poco frequenti, mentre la reattività del sistema è fondamentale;
- Sistemi per workstation → la gestione delle richieste di memoria è fondamentale, come nel caso dei grandi server di basi di dati.

Il supporto teorico alla soluzione del problema è scarso e la progettazione di un buon PFRA è un task basato prettamente su conoscenze empiriche.

La diretta conseguenza è che la parte di codice che implementa questo componente varia molto velocemente nel tempo.

IL PFRA di Linux si basa su 4 scelte euristiche fondamentali:

- Seleziona prima le pagine *harmless*
 - Si tratta di quelle pagine in qualche cache e che non sono referenziate da alcun processo. Esse vanno selezionate prima delle pagine associate allo spazio di indirizzi di qualche processo;

- Rendi tutte le pagine di un processo in User Mode reclamabili
 - Eccezione fatta per le pagine locked, tutte le pagine di un processo sono potenzialmente soggette a reclaiming. Se un processo è in sleep per lungo tempo, potrebbe perdere tutte le sue pagine;
- Reclama le pagine condivise dopo aver eliminato i collegamenti nelle page table;
- Reclama solo le pagine non utilizzate (sfrutta il LRU)

8.21 Periodic Reclaiming

Il PFRA esegue periodicamente un reclaiming, attraverso 2 diversi meccanismi:

- Il kernel thread kswapd
 - Necessario poichè alcune richieste di allocazione sono effettuate da interrupt e deferrable function, che non permettono la sospensione del processo
 - I kernel thread possono sfruttare cicli di clock, in cui la CPU sarebbe idle;

Quando la funzione `__alloc_pages()` si rende conto che le pagine di memoria sono sotto una soglia critica, sveglia il thread per il corrispondente nodo di memoria.

La funzione `cache_reap()` → questa funzione libera pagine allocate a oggetti liberi nelle lastre dello slab allocator. Ogni volta che viene eseguita, la funzione richiede allo scheduler di pianificare la sua successiva esecuzione

9 Lezione del 31-03

9.1 Virtual File System (VFS)

I sistemi Linux presentano la capacità di coesistere e interagire con altri sistemi presenti sulla stessa macchina.

Un esempio concreto di tale capacità è dato dalla possibilità di montare dischi o partizioni, che presentano formati di file differenti.

Ciò è possibile attraverso un SW layer del kernel che prende il nome di Virtual File System che gestisce l'intero insieme di system call tipico dei filesystem standard UNIX.

Il VFS lavora come livello di astrazioni fra i programmi di applicazione e la reale implementazione di un filesystem

Divisi in 3 categorie:

- Disk-based filesystem → gestiscono le informazioni memorizzate in dischi locali o in dispositivi, il cui comportamento è assimilabile a quello di un disco locale;
- Network filesystem → permettono l'accesso alle informazioni memorizzate su pc collegati in rete;
- Special filesystem → non gestiscono dati realmente memorizzati su un disco locale o remoto;

9.1.1 Common File Model

Rappresenta e modella tutti i filesystem supportati. Nel CFM tutte le entità coinvolte sono rappresentate attraverso file.

Dato che Linux non implementa i filesystem nel loro formato nativo, deve ricostruire tali oggetti in memoria "on the fly". Si può guardare al CFM come ad un sistema ad oggetti, caratterizzati da dati, che conservano le informazioni e metodi che servono a manipolarle.

Un CFM consiste in:

- Il **superblocco** → contiene informazioni relative al filesystem che è stato montato;
- L' **inode** → contiene informazioni relative ad un file specifico
 - Ciascun oggetto di tipo inode è associato ad un inode number, che identifica in maniera univoca il file nel filesystem;
- Il **file object** → contiene informazioni sul tipo ed il grado di interazione fra un processo ed un file aperto
 - Queste informazioni sono tenute solo in memoria centrale e non trovano alcuna corrispondenza con alcun elemento sul disco;
- Il **dentry object** → contiene le informazioni che associano ciascuna entry della directory con il file corrispondente
 - Generalmente ogni filesystem ha un modo peculiare di memorizzare questa associazione.

9.1.2 Strutture dati del VFS

Il VFS è contenuto in un oggetto, implementato da una struttura dati **super_block**, la quale contiene una serie di campi dati ed un puntatore ad una tabella di funzioni, che rappresentano i metodi dell'oggetto.

9.1.3 L'oggetto superblock

Tutti gli oggetti superblock sono collegati tra loro in una lista doppiamente concatenata la cui testa è contenuta nella variabile **super_blocks**, mentre il puntatore **s_list** contiene l'indirizzo del successore.

Il campo **s_fs_info** punta ad una struttura dati contenente le informazioni sul tipo di filesystem, cui il superblock fa riferimento.

Per questioni di efficienza, Linux mantiene e aggiorna una copia dell'oggetto superblock in memoria, impostando un flag **s_dirt** per assicurarne la sincronizzazione con la copia su disco

9.1.4 Operazioni sul superblock

La struttura `super_block` contiene un campo `s_op`, che punta ad una struttura `super_operations` relativa ai metodi implementati per tale oggetto.

<code>alloc_inode()</code>
<code>destroy_inode()</code>
<code>read_inode()</code>
<code>dirty_inode()</code>
<code>write_inode()</code>
<code>delete_inode()</code>
...
<code>quota_read()</code>
<code>quota_write()</code>

Queste operazioni sono disponibili per qualunque file system. Se i metodi non sono implementati, i corrispondenti puntatori sono NULL.

9.1.5 L'oggetto inode

Tutte le informazioni di cui il filesystem ha bisogno di gestire un file, sono contenute nella struttura inode. Sebbene il nome di un file possa cambiare, l'inode che gli è stato associato rimane sempre lo stesso.

Linux mantiene una copia di un inode in memoria e usa un flag `i_state` per capire se è necessario sincronizzarne il contenuto con la sua copia su disco.

Il campo `i_state` fornisce anche informazioni quali:

- `I_LOCK` (operazioni di I/O);
- `I_FREEING` (rilasciato);
- `I_CLEAR` (contenuto non più significativo);
- `I_NEW` (allocato e non ancora utilizzato);

9.1.6 Il File Object

Describe il modo in cui un processo interagisce con un file che ha aperto. L'oggetto viene creato all'apertura del file e consiste in una struttura, che non ha un corrispondente su disco, ed è quindi priva di un flag `dirty`.

L'informazione più importante memorizzata all'interno della struttura è il **file pointer**. Esso è conservato qui e non nell'inode, poiché più processi potrebbero accedere allo stesso file.

I file object sono allocati in oggetti detti **filp**, conservati insieme al loro descrittore, in una delle cache dello slab allocator

9.1.7 Dentry File

Quando una directory viene letta in memoria, il VFS la trasforma in un oggetto dentry, implementato dalla struttura dentry. Il kernel scomponete un path nelle sue componenti e crea un oggetto dentry per ciascuna di esse. Questo oggetto associa ciascuna parte al corrispondente inode.

9.1.8 Tipi di filesystem

Linux offre la possibilità di gestire diversi tipi di filesystem, di conseguenza il kernel deve avere a disposizione tutte le informazioni necessarie a poterli gestire.

Il kernel rende possibile questa opzione attraverso una procedura, che va sotto il nome di **Filesystem Type Registration**.

Ciascun filesystem è caratterizzato da una struttura dati di tipo **file_system_type**. Tutti i filesystem object sono contenuti in una lista concatenata.

9.1.9 La gestione dei filesystem

All'avvio del sistema, la funzione **register_filesystem()** inserisce un oggetto di tipo **file_system_type** per ciascun tipo di filesystem disponibile all'interno della lista dei tipi di filesystem.

Questa funzione viene invocata anche quando il kernel carica un modulo che implementa un filesystem.

Inoltre è disponibile una funzione **get_fs_type()**, che prende in input il nome di un filesystem e restituisce il corrispondente **file_system_type** object.

Il Filesystem principale (root) è il primo ad essere caricato ed è quello in cui sono presenti i programmi e gli script necessari all'avvio del sistema.

I filesystem possono essere montati su una directory di un filesystem già montato, che viene considerata come **mounting point** e diviene la root del filesystem montato.

Il nuovo filesystem viene considerato come **child** del filesystem sul quale è stato montato.

La root directory di un filesystem appena montato, nasconde il contenuto della directory usata come mount point.

9.1.10 I namespace

I filesystem sono organizzati come un albero e nei sistemi Unix tradizionali esiste sempre un solo albero.

Linux, si differenzia in questo, facendo uso dei **namespace**.

Diversi processi possono condividere lo stesso namespace, ossia l'albero dei filesystem correntemente montati. Tuttavia, se un processo monta un nuovo filesystem, questo rimane visibile solo a questo, modificando solo il namespace del processo che lo ha montato.

Il namespace è implementato da una struttura dati chiamata namespace, il cui puntatore è un campo del process descriptor

9.1.11 Filesystem Mounting

Nei sistemi Unix tradizionali, ciascun Filesystem può essere montato solo una volta. Linux differisce in questo → è possibile montare lo stesso filesystem più volte, su directory differenti.

Sebbene lo stesso filesystem possa essere acceduto da diversi punti, esso rimane unico, ossia **esiste un solo superblocco** nel sistema che lo identifica.

È possibile montare più filesystem differenti sullo stesso mount point; tuttavia, ciascun nuovo filesystem nasconde il vecchio

10 Lezione del 09-04

Questo tipo di sistema operativo deve garantire che una elaborazione (o task) termini entro un dato vincolo temporale o scadenza (detta in gergo deadline). Per garantire questo è richiesto che la schedulazione delle operazioni sia fattibile. Il concetto di fattibilità di schedulazione è alla base della teoria dei sistemi real-time ed è quello che ci permette di dire se un insieme di task sia eseguibile o meno in funzione dei vincoli temporali dati.

Questo tipo di sistema è implementato via SW.

Il collante tra il livello SW che implementa la logica del sistema di controllo e le risorse HW controllate è il SO.

È presente quindi un livello di astrazione **HAL** (Hardware Abstraction Level).

I task non real-time vengono gestiti dal HAL con una politica **BEST EFFORT**. Quelli real-time vengono gestiti attraverso lo **SCHEDULER** che implementa algoritmi di scheduling specifici

10.1 Event Driven

Un SO di questo tipo è in grado di schedulare un task nello stesso istante in cui si verifica l'evento che lo ha attivato



Vantaggi	Svantaggi
Intuitivo e semplice da usare	Complesso da realizzare

10.2 Time Driven

Un SO event-driven è irrealizzabile se l'unità di elaborazione è di tipo digitale e quindi intrinsecamente quantizzato nel tempo

Un approccio realizzabile consiste nel rilevare periodicamente l'occorrenza di eventi e di gestire di conseguenza i relativi task. Tale approccio prende il nome di *Time Driven*. Il periodo di tempo che intercorre tra 2 rilevazioni consecutive è detto **periodo di rilevazione** (o periodo di scansione) T_s .

Deve gestire necessariamente le problematiche relative alla gestione sincrona di eventi asincroni (che attivano i task). In un sistema di controllo digitale, un evento può essere associato al valore logico di una variabile binaria (segnale logico).

Vantaggi	Svantaggi
Semplice da realizzare e reattivo	Flessibilità

10.2.1 Problema 1 → Osservabilità degli eventi

Un evento può non essere osservabile. Il segnale logico associato all'evento rimane attivo per un tempo inferiore del periodo di rilevazione.

10.2.2 Problema 2 → Ritardo di rilevazione

Ogni occorrenza di un task è soggetta ad un ritardo di rilevazione che impatta necessariamente sullo start time del task.

10.2.3 Problema 3 → Ordine di occorrenza

L'ordine in cui si presentano 2 o più eventi occorrenti tra due rilevazione successive viene perso.

10.3 WatchDog

Il So controlla periodicamente che le deadline dei task real-time vengano rispettate attraverso un **WATCHDOG TIMER**. Allo scadere del timer se una deadline è scaduta, l'anomalia è segnalata e viene eseguita una routine di emergenza

10.4 Task periodici e Aperiodici

I task che devono essere eseguiti ogni p unità di tempo sono detti periodici (di periodo p).

Tutti gli altri task sono aperiodici

Task che richiedono il processore in modo impredicibile sono detti **sporadici**

10.5 Deadline

Istante di tempo dopo il quale la computazione non è semplicemente in ritardo, ma errata.

Si tratta di vincoli di tempo stringenti imposti ai task dall'ambiente esterno

10.6 Scheduling Real-Time

- Hard Real-Time → scadenza obbligatoria
 - In caso contrario si potrebbe incorrere in malfunzionamenti;
- Soft Real-Time → scadenza desiderabile ma non obbligatoria;

Gli eventi legati all'esecuzione di tali task possono essere o periodici o aperiodici.

Ciò implica che un sistema che deve eseguire diversi task periodici è in grado di gestirli solo se i tempi da essi richiesti soddisfano determinati vincoli

Dati m task periodici con un periodo P_i e richiedenti C_i secondi di tempo CPU per essere gestiti, essi possono essere soddisfatti solo se:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

10.7 Rate Monothonic Scheduler

Scheduler a priorità statica (abbreviato RMS).

Assegna le priorità ai task in base al loro periodo T, inteso come la quantità di tempo compresa fra l'arrivo di un istanza del task e la successiva.

Il tempo di esecuzione C è la quantità di tempo di elaborazione richiesta da ogni occorrenza del task e in un sistema single-core il tempo di esecuzione non deve essere maggiore del periodo, ossia $C < T$

Se tutti i processi vengono eseguiti completamente, si può calcolare l'utilizzo del processore come

$$U = \frac{C}{T}$$

10.7.1 Test di schedulabilità

Condizione sufficiente, ma non necessaria data da:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

Se il numero di processi n tende ad infinito, questa espressione tende a

$$\lim_{n \rightarrow \infty} n(\sqrt[n]{2} - 1) = \ln(2)$$

RMS può garantire tutte le scadenze con un utilizzo del processore intorno al 69,3% e quindi il restante 30,7% della CPU può essere utilizzato per la schedulazione di processi non real-time

10.8 EDF Scheduler

Utilizza una coda di priorità e tiene in considerazione le scadenze di ogni task. Ogni volta che viene chiamata la funzione di schedulazione la coda viene analizzata alla ricerca del processo più vicino alla sua scadenza, il quale viene schedulato per l'esecuzione.

Particolarmente ottimo per i sistemi single-core con pre-rilascio.

Nel caso in cui ci siano processi periodici, anch'essi con delle scadenze (uguali al loro periodo) EDF ha un utilizzo della CPU vicino al 100%. Il test di schedulabilità per EDF è:

$$U = \sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Limiti di questo algoritmo sono:

- Se il sistema è sovraccarico è impossibile prevedere il numero di processi che non rispettano la scadenza;
- Algoritmo difficile da implementare, per via di come dover rappresentare le scadenze

10.9 Sistemi Soft e Hard Real-Time

Un task real-time può essere distinto in 3 categorie:

- Hard → se la mancata scadenza può causare conseguenze catastrofiche sul sistema sotto controllo;
- Firm → se la produzione dei risultati dopo la scadenza è inutile per il sistema, ma non causa alcun danno;
- Soft → se la produzione dei risultati dopo la scadenza ha ancora qualche utilità per il sistema, sebbene causi un peggioramento delle prestazioni.

10.9.1 Gestione di task in contesti ibridi

Le applicazioni del mondo reale includono hard, firm e soft task, per cui un sistema real-time deve essere progettato per gestire i diversi tipi di task implementando strategie diverse:

- **Hard** → devono essere garantite offline;
- **Soft** → devono essere garantite online. interrompendoli se la loro scadenza non può essere rispettata;
- **Soft** → devono essere gestiti con l'obiettivo di ridurre al minimo il tempo di risposta medio.

10.9.2 Limiti dei comuni sistemi Real-Time

La gran parte dei sistemi real-time adottati per il controllo dei sistemi è basata su kernel, che sono riadattamenti degli analoghi utilizzati in sistemi time-sharing, con conseguenti svantaggi:

- Multitasking → gestita mediante system call, che non tengono esplicitamente considerazione del tempo, provocando ritardi imprevedibili;
- Scheduling basato su priorità → molto flessibili, ma hanno un numero limitato di livelli di priorità e non sempre i requisiti di tempo sono mappabili su tali intervalli. Inoltre, l'arrivo di un nuovo task potrebbe richiedere il ricalcolo di tutte le priorità;
- Reattività agli interrupt esterni → agli interrupt esterni è assegnata una priorità più alta di quella dei task da eseguire, che potrebbero rimanere sospesi per un tempo imprevedibile;
- IPC e sincronizzazione → i semafori binari normalmente usati provocano fenomeni quali inversione della priorità, attese a catena e deadlock, che rendono imprevedibile la durata dei task;
- Kernel ridotti e contest switch veloci → benché questi riducano l'overhead del sistema non garantiscono esplicitamente il rispetto della deadline;
- Assenza di un real-time clock → non offrono la possibilità di gestire in modo esplicito il timing dei task.

10.9.3 Desiderata per i Sistemi Real-Time

Applicazioni di controllo complesse che richiedono forti vincoli temporali nell'esecuzione dei task devono essere supportati da sistemi operativi altamente prevedibili.

La prevedibilità può essere ottenuta solo introducendo cambiamenti radicali nei paradigmi di progettazione di base dei sistemi classici timesharing.

Il sistema è in grado di verificarne la schedulabilità, avendo come vincolo il rispetto delle deadline piuttosto che la riduzione del tempo medio di risposta.

Fra i desiderata per un SO real-time si annoverano:

- Timeliness → validità del risultato è intesa sia in termini di valore, che di tempo entro il quale esso è fornito;
- Prevedibilità → il SO deve essere in grado di prevedere le conseguenze di qualunque scelta nello scheduling;
- Efficienza → questo tipo di sistemi generalmente gira su HW limitati;
- Robustezza → non devono colllassare in caso di eccessi di carico;
- Fault Tolerance → problemi HW o SW non devono mandare in crash l'intero sistema;
- Manutenibilità → devono essere progettati secondo uno schema modulare in modo da semplificare modifiche e integrazioni.

10.9.4 Prevedibilità

Il più importante tra i requisiti di un sistema hard real-time. Influenzata da fattori HW quali:

- DMA;
- Cache;
- Interrupts;
- System calls;
- Semaphores;
- Memory Management;
- Programming Language.

Gli Interrupts

Rappresentano un grosso problema per la prevedibilità di un sistema real-time, perché, se non gestiti correttamente, possono introdurre ritardi notevoli nell'esecuzione di un task. Distinguiamo 3 approcci diversi:

- A → tutti gli interrupt esterni sono disabilitati a meno del timer
 - Le operazioni di I/O sono gestite direttamente dal processo mediante polling;
- B → tutti gli interrupt esterni sono disabilitati a meno del timer
 - Le operazioni di I/O sono gestite da routine del kernel attivate periodicamente dal timer;

- C → tutti gli interrupt esterni sono abilitati, ma i driver hanno un ruolo minimale
 - L'unica mansione di un driver è attivare un task il cui scopo è gestire il dispositivo;
 - La priorità di questi task è inferiore rispetto a quelli con deadline stringenti

I semafori

Questo meccanismo non è adatto a sistemi in tempo reale perchè soggetto al fenomeno di inversione di priorità, che si verifica quando un'attività ad alta priorità viene bloccata da un'attività a bassa priorità per un intervallo di tempo illimitato. Varie soluzioni efficaci a questo problema:

- Basic Priority Inheritance;
- Priority Ceiling;
- Stack Resource Policy.

10.10 I linguaggi di programmazione

Gli attuali linguaggi di programmazione non sono abbastanza espressivi da caratterizzare determinati comportamenti temporali e, quindi, non sono adatti per realizzare applicazioni prevedibili in tempo reale.

Sono stati proposti nuovi linguaggi ad alto livello per supportare lo sviluppo di applicazioni hard real-time:

- Real-Time Euclid;
- Real-Time Concurrent C.

10.10.1 Real-Time Euclid

Impone al programmatore di specificare limiti temporali ed eccezioni di timeout in tutti i loop, le attese e le istruzioni di accesso ai dispositivi. Impone altre restrizioni quali:

- Assenza di strutture dinamiche → renderebbero imprevedibili i tempi di allocazione e deallocazione;
- Assenza di ricorsione → questa impedirebbe la stima del tempo di esecuzione delle chiamate ricorsive nell'analisi della schedulabilità;
- Cicli temporizzati → il programmatore deve specificare il numero massimo di iterazioni in ogni ciclo al fine di rendere possibilie la stima della durata di un task.

10.10.2 Real-Time Concurrent C

Estende il concurrent C fornendo delle facility per indicare la **periodicità/aperiodicità** e le deadline. Fornisce costrutti del tipo:

within deadline (d) statement-1

[else statement-2]

Costrutti quindi che permettono di specificare le istruzioni da eseguire nel caso in cui la deadline sia (non sia) rispettata

A differenza di Real-Time Euclid, che è stato progettato per supportare la verifica statica di sistemi real-time, Real-Time Concurrent C è orientato a sistemi dinamici, in cui è possibile attivare task in fase di esecuzione.

10.11 Sistemi Embedded

10.11.1 Caso Semplice I

Unico task periodico ripetuto ciclicamente.

All'accensione il sistema esegue direttamente l'algoritmo di controllo, scritto a partire dalla prima posizione in memoria (ROM) che consiste in un ciclo perpetuo che esegue il task.

Le deadline sono verificate se l'esecuzione del task è sufficientemente veloce

10.11.2 Caso Semplice II

Unico task periodico, invocato a intervalli regolari da un timer.

Il timer lancia un'interruzione, il task è realizzato dalla routine di servizio per l'interruzione relativa, la CPU altrimenti è in IDLE.

Le deadline sono verificate se l'esecuzione del task è sufficientemente veloce, mentre l'interruzione non interrompe la normale esecuzione.

10.11.3 Caso Complesso I

Più task periodici, di ugual periodo.

Ad ogni interruzione del timer vengono eseguiti tutti i task.

Le deadline sono verificate se la somma dei tempi di esecuzione dei task è inferiore alla deadline.

Se i periodi non sono uguali, ma multipli tra loro, le interruzioni seguono il periodo minore e ad ogni interruzione si sceglie quali task eseguire.

10.11.4 Caso Complesso II

Più task periodici, di periodo diverso, e task sporadici.

Le interruzioni dal timer arrivano secondo i vari periodo, e vi sono anche le interruzioni relative agli eventi che scatenano i task sporadici. Ogni interruzione è gestita da una propria routine che esegue il task relativo.

La verifica della deadline diventa difficile.

10.11.5 Arduino

Strumento open source che semplifica la progettazione e la prototipazione elettronica.

La scheda arduino presenta un HW molto modificabile, riprogettabile ed estensibili senza alcun limite (si sfruttano le librerie C++).

10.11.6 Microcontroller

Dispositivo elettronico integrato su un unico chip, progettato appositamente per interagire con input esterni, analogici o digitali, e restituire output analogici o digitali

10.11.7 La scheda Arduino

Ogni scheda presenta un ulteriore chip per la conversione del segnale digitale da USB a seriale.

Tramite il SW di Arduini è possibile programmare i pin come porte di entrata o di uscita digitale o analogica e se ne può controllare il comportamento

10.11.8 Programmare in Arduini

Comporta sostanzialmente scrivere due funzioni fondamentali:

- `setup();`
- `loop();`

10.12 RT Linux

SO nel quale un piccolo kernel real-time coesiste con il kernel di Linux (like-POSIX).

L'intenzione è quella di fare uso dei servizi sofisticati ed altamente ottimizzati (per quel che concerne il comportamento medio) di un sistema di computer standard time-shared, mentre ancora si permette alle funzioni real-time di operare in un ambiente prevedibile ed a bassa latenza.

In questo SO un task real-time è eseguito su un kernel non real-time, che è concepito come un task a più bassa priorità, utilizzando un livello di VM per rendere il kernel standard completamente prelazionabile.

10.13 RTAI

Versione di Linux in cui sono state introdotte una serie di regole, che limitano il potenziale del Linux standard ad agire come un SO real-time.

11 Lezione del 16-04

11.1 Virtualizzazione

È possibile eseguire uno o più SO da un unico PC, in un ambiente protetto e monitorato che prende il nome di macchina virtuale:

- Il SO in cui viene eseguita la macchina virtuale è detto *host*;
- La macchina virtuale che viene creata è detta *guest*.

Il codice della VM viene eseguito direttamente dal SO ospitante, ma il sistema ospite *pensa* di essere eseguito su una macchina reale priva di emulazione o virtualizzazione HW

11.2 Macchina virtuale - Definizione

Una macchina che esiste al di sopra della macchina reale.

Questa non esiste fisicamente, ma realizzata mediante SW. L'utente interagisce con la macchina grazie ad un opportuno linguaggio che la macchina tradurrà in comandi per l'HW sottostante.

Il SW di base fornisce un insieme finito di comandi (linguaggio comandi) che la macchina è in grado di eseguire.

Possono esistere un numero indefinito di VM che possono essere create. Il SW di base moderno può essere visto come una gerarchia di VM, organizzate a *cipolla*. Ognuno dei livelli fornisce un insieme di funzioni che diventano sempre più astratte man mano che ci si allontana dalla macchina fisica.

11.3 Definizione della virtualizzazione

Virtualizzare il sistema significa presentare all'utilizzatore una visione delle risorse del sistema diversa da quella reale. Questo si ottiene introducendo un **livello di indirezione** tra la vista logica e la vista fisica delle risorse. L'obiettivo è disaccoppiare il comportamento delle risorse HW e SW di un sistema di elaborazione, così come viste dall'utente, dalla loro realizzazione fisica

es:

- Astrazione;
- Linguaggio di Programmazione;

- A livello di processo.

La comunicazione con le macchine guest attraverso un componente SW di livello intermedio generalmente denominata *hypervisor* o *VMM* (virtual machine monitor).

11.4 Vantaggi

- Uno sviluppatore di SW può eseguire la sua applicazioni in diversi ambienti sulla stessa macchina;
- Un amministratore di sistemi può testare uno scenario complesso che veda interagire più servizi su host diversi, ricreandolo su più VM ospitate su una singola macchina fisica.

Un VMM è il mediatore unico nelle interazioni tra VM e l'HW sottostante che garantisce:

- Isolamento tra le VM;
- Stabilità del sistema.

11.5 Tipi di Virtualizzazione

- Emulation;
- Full virtualization;
- Paravirtualization;
- Operating system level virtualization.

11.5.1 Emulation

L'HW viene completamente emulato dal programma di controllo. Risulta lento a causa della traduzione delle istruzioni dal formato del sistema ospite a quello ospitante.

L'SW di virtualizzazione si incarica di presentare al SO guest un'architettura HW completa a lui nota, indipendentemente dall'architettura HW presente sulla macchina host.



L'approccio dell'emulazione sono state:

- **Interpretazione**

- Lettura di ogni singola istruzione del codice che deve essere eseguito e sulla esecuzione di più istruzioni sull'host virtualizzante per ottenere semanticamente lo stesso risultato;
- Molto generale e potente;
- Produce un grande sovraccarico.

- **Compilazione dinamica**

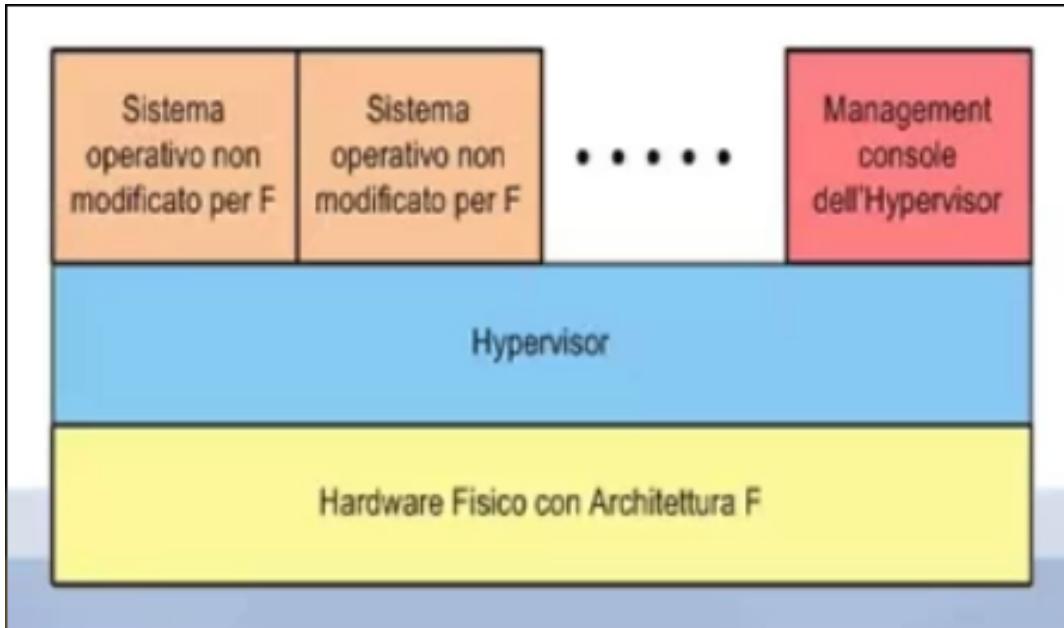
- Invece di leggere una singola istruzione, vengono letti interi blocchi, che vengono analizzati e tradotti per la nuova architettura ottimizzandoli e messi in esecuzione;
- Il codice può essere bufferizzato;
- Maggiore ottimizzazione e traduzione.

Limiti

La gestione della CPU consiste nell'interpretare il flusso di istruzioni dal sistema guest ed eseguire sul processore host una serie di istruzioni semanticamente equivalenti. Questo può essere reso possibile con l'assegnazione di una variabile per ogni registro e flag della CPU simulata → risulta una tecnica di virtualizzazione inefficiente.

11.5.2 Full Virtualization

L'hypervisor implementa l'isolamento necessario a separare il sistema ospite dall'HW fisico della macchina (es di SW che utilizza questa modalità è VirtualBox).



Con questo tipo di virtualizzazione (detto Native Virtualization), i SO guest girano senza alcuna modifica in VM ospitante sul sistema host. A differenza dell'emulazione, richiede che i sistemi guest siano compatibili con l'architettura HW della macchina fisica.

La compatibilità permette che molte istruzioni possano essere eseguite direttamente sull'HW senza bisogno di un SW di traduzione.

11.5.3 VMM di sistema - realizzazione

L'architettura della CPU prevede, almeno 2 livelli di protezione → **supervisore** e **utente**. Solo il VMM opera nel primo stato, mentre il SO e le applicazioni operano nel secondo stato. Sono presenti 2 problemi:

- **Ring deprivilegging** → il SO della VM esegue in uno stato che non gli è proprio;
- **Ring compression** → la separazione tra gli stati non è rispettato (applicazioni e SO della macchina eseguono allo stesso livello).

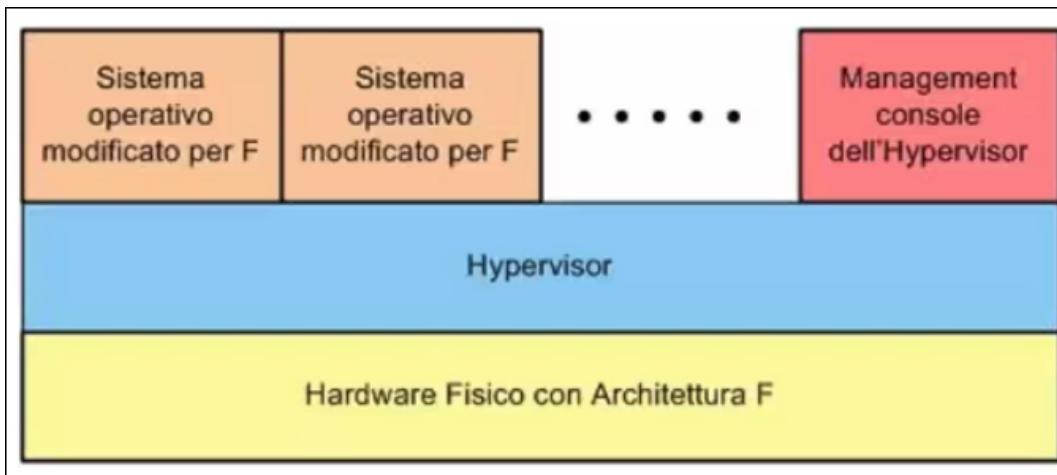
Una possibile soluzione potrebbe essere la seguente:

1. Se il guest tenta di eseguire un'operazione privilegiata, la CPU notifica un'eccezione al VMM e gli trasferisce il controllo (trap) → il VMM controlla la correttezza dell'operazione richiesta e ne emula il comportamento
 - (a) le istruzioni non privilegiate possono essere eseguite direttamente dall'HW senza alcun intervento da parte della CPU (esecuzione diretta);

11.5.4 Paravirtualization

Il programma di controllo fornisce un'API (interfacce di programmazione applicativa) per l'hypervisor, che viene utilizzata dal sistema ospite per interagire con l'HW. Un esempio è il KVM.

L'hypervisor presenta alle VM una versione modificata dell'HW sottostante, mantenendo tuttavia la medesima architettura. Il SO delle VM è modificato per evitare alcune particolari chiamate di sistema.



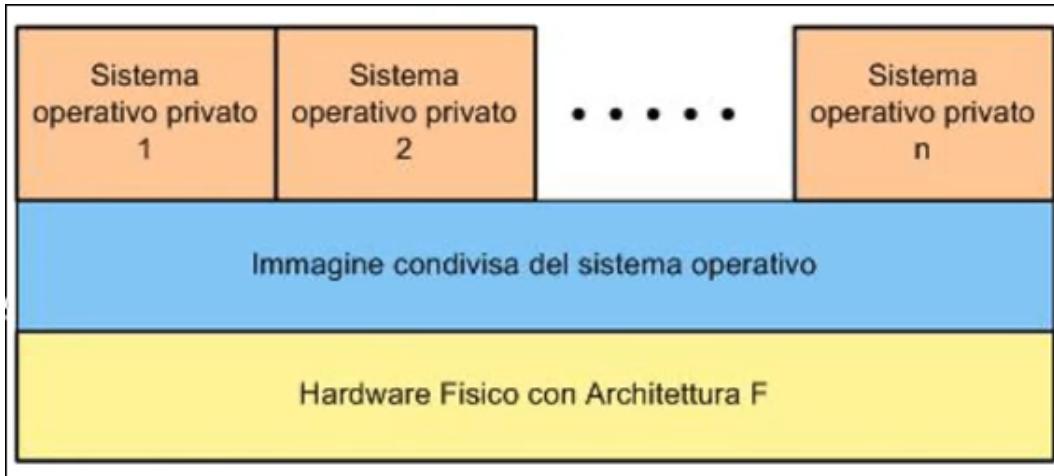
Questa tecnica permette di ottenere un decadimento delle prestazioni minimo rispetto al SO non virtualizzato. Le istruzioni provenienti dalle VM vengono eseguite quasi tutte direttamente sul processore senza che intervengano modifiche.

11.5.5 Operating System level Virtualization

Non si utilizza un hypervisor, ma la virtualizzazione è creata usando copie del SO installato sull'host. I sistemi guest creati sono istanze del SO host con un proprio file system, configurazione di rete e applicazione.

Il vantaggio è il miglior utilizzo delle risorse grazie alla condivisione di spazi di memoria.

LE istanze guest non richiedono un kernel privato, ma utilizzano lo stesso con un conseguente minor utilizzo di memoria fisica



11.6 Architettura VMware

VMware è una delle piattaforme di virtualizzazione leader di mercato.

Sfrutta come hypervisor **ESX Server** (direttamente installato su un server fisico → in questo caso si usa spesso il termine **bare metal**)

Esso ha accesso diretto alle risorse HW del server e completo controllo su CPU, memoria, dischi e rete. I principali componenti logici su cui si basa l'architettura di ESX Server sono:

- Virtualization layer;
- Resource manager;
- HW Interface component;
- Service console.

11.7 Virtualizzazione delle risorse

Tecnologie come cluster, grid e cloud computing hanno tutte lo scopo di consentire l'accesso a grandi quantità di potenza di calcolo in modo completamente virtualizzato, aggregando le risorse e offrendo un'unica vista di sistema. L'utility computing descrive un modello di business per la fornitura di potenza di calcolo su richiesta. I consumatori pagano i fornitori in base all'utilizzo (pay as-you-go).

11.8 Cloud computing

Modello pay-per-use per abilitare in modo conveniente e on-demand l'accesso alla rete disponibile a un pool condiviso di risorse di calcolo configurabili che possono essere rapidamente fornite e rilasciate con il minimo sforzo di gestione o interazione con il fornitore di servizi. Le sue radici derivano dal progresso delle seguenti tecnologie:

- HW;

- Tecnologie internet;
- Calcolo distribuiti;
- Gestione dei sistemi.

11.8.1 I precursori - Grid Computing

Si è immaginato un'infrastruttura computazione in grado di far leva sulla potenza di calcolo inutilizzato delle macchine in stato IDLE connesse a internet.

Caratterizzato da 2 aspetti fondamentali:

- Utilizzo della Virtualizzazione dell'HW come base per l'infrastruttura di calcolo;
- Utilizzo di tecnologie Web based per l'accesso a un'infrastruttura di calcolo distribuita.

Il Grid computing sfrutta ampiamente i Web Services sulla base del modello SOA, che un ruolo chiave nell'accesso alle griglie computazionali.

Il cloud computing riporta a un'idea di macro produttori di computing indipendentemente dal tipo di cloud.

La differenza nel tempo è variata → il grid computing ha prodotto l'aggregazione di centri di supercalcolo che oggi sono l'ossatura del cloud computing

Desktop Grid → PC di utenti normali che vengono utilizzati per la loro potenza di calcolo e inglobati in un grid computing.

11.8.2 I precursori - EDGeS

Progetto che raccoglie più di 50 nazioni con il fine comune di costruire un'infrastruttura Grid caratterizzato da:

- Uso delle più avanzate e recenti tecnologie Grid;
- Servizi disponibili indipendentemente dalla locazione geografica degli utenti;
- Interesse nell'attrarre un ampio spettro di nuovi utenti ed utilizzi.

11.8.3 Service-Oriented Architecture (SOA)

Paradigma per l'organizzazione e l'utilizzo delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti.

12 Lezione del 19-04

12.1 Tipologie di Grid

Si pensa al Grid computing come una utility on-demand → Utility Computing. Attualmente esistono 2 tipi di Grid:

- Orientate all'High Performance Computing (HPC);
- Orientate alla sperimentazione SW che generalmente dà comunque supporto all'HPC.

Queste 2 tipologie di Grid hanno in comune l'aggregazione di risorse di supercalcolo costituite da computational cluster appartenente a domini amministrativi diversi.

12.2 Dal Grid al Cloud: l'evoluzione

La potenza di calcolo non costituisce il target strutturale del Cloud computing, ma è dettata dalla necessità di offrire un servizio di elevata scalabilità in grado di assorbire richieste da un sempre crescente numero di utenti.

12.3 Cloud Vs Grid: la rivoluzione

Il termine Cloud è volutamente vago e sottintende una tipologia di organizzazione dall'infrastruttura di calcolo slegata teoricamente da qualsiasi vincolo tecnologico. Il cloud può essere considerato rivoluzionario poichè mira a vendere distributed computing. Il modello è quello delle utility di larga scala → l'incremento del numero di utente ha un costo quasi nullo ma moltiplica i profitti e al contempo contribuisce alla riduzione dei prezzi.

12.4 Cloud Computing

Insieme di tecnologie informatiche che permettono di fornire come servizi on-demand l'accesso e le risorse di elaborazione, di memoria ed applicative, distribuite e virtualizzate in rete:

- Server;
- Storage.

Tipicamente l'utente finale accede a tali servizi mediante un browser

12.5 Infrastrutture IT onsite (senza cloud)

Vantaggi	Svantaggi
Maggiore controllo (tutto in loco)	Richiede un grande capitale iniziale
Maggiore privacy (tutto in loco)	Richiede costi di manutenzione continui
	La tecnologia deve essere sostituita periodicamente
	Richiede personale IT interno da mantenere

12.6 Cloud IT infrastructure

Vantaggi	Svantaggi
Convenienza	Meno controllo sull'ambiente
Migliore accessibilità da dispositivi mobili	Necessita di una connessione internet di qualità
Servizi di qualità superiore	
Licenze SW non richieste	

12.7 Thin client

Macchine con limitate capacità e prestazioni, ma in grado di accedere efficacemente ad Internet per usufruire dei servizi in rete.

Ciò porta a diverse conseguenze:

- Lock-in degli utenti su una specifica piattaforma proprietaria;
- Perdita di controllo sui dati sensibili aziendali memorizzati nell'infrastruttura di un fornitore di Cloud Computing;
- Concentrazione dei servizi di Cloud Computing nelle mani di pochi grossi player;
- Danni dovuti a disservizi su larga scala.

12.8 Riferimenti

Un cloud a riferimento a un ambiente IT progettato per il provisioning remoto di risorse IT scalabili e misurate. Bisogna specificare la differenza tra "cloud" e il simbolo cloud da Internet. Come ambiente specifico utilizzato per

fornire in remoto risorse IT, un cloud ha un confine definito. Ci sono molte nuvole individuali che sono accessibili via Internet

Un'altra distinzione fondamentale è che non è necessario per i cloud essere basati sul Web anche se sono comunamente basati su protocolli e tecnologie internet

12.9 Risorse

Elemento fisico o virtuale correlato all'IT che può essere basato su SW, come un server virtuale o un programma SW personalizzato o basato su HW, come un server fisico o un dispositivo di rete

12.10 On Premise

È un modo di affermare "nei locali di un ambiente IT controllato che non è basato sul cloud"

Una risorsa IT che è on-premise non può essere basata su cloud e viceversa.

12.11 Scaling

Il ridimensionamento, rappresenta la capacità della risorsa IT di gestire richieste di utilizzo aumentate o ridotte. Esistono 2 tipi di scaling:

- Horizontal Scaling → Scaling out and scaling in;
- Vertical Scaling → Scaling up and scaling down;

Entrambi i tipi presentano vantaggi e svantaggi (molto spesso ci si trova in sistemi dove sono presenti entrambi i tipi)

Horizontal	Vertical
Meno costosi	Più costosi
Risorse IT disponibili immediatamente	Risorse IT disponibili immediatamente
Replicazione delle risorse e scaling automatico	Richiesto un setup aggiuntivo
Non limitato dall'HW	Limitato dall'HW
Richieste di risorse IT aggiuntive	Nessuna richiesta di risorse IT aggiuntive

12.12 Filoni Tecnologici nel Cloud computing

- API per l'esposizione e la composizione di funzionalità distribuite;

- Meccanismi per le chiamate remote;
- Linguaggi di programmazione con prevalenza di quelli di alto livello;
- Soluzioni per la rappresentazione e la condivisione dei dati (XML o JSON);
- Meccanismi per il riperimento e la memorizzazione di dati;
- Meccanismi di monitoraggio e di accounting delle risorse utilizzate

12.13 Conoscenze informatiche che contribuiscono al Cloud Computing

- Application server → responsabili di funzioni, quali la presentation, il business e il data logic, con la differenza che esse sono ora distribuite su sistemi cooperanti;
- Data management → sfrutta le funzioni dei search engine per identificare, indicizzare e memorizzare le informazioni
 - Queste sono in fase di ulteriore sviluppo per integrare capacità semantiche e per favorire il data mining intelligente;
- Rich Internet Application (RIA) → forniscono le tecnologie per l'accesso ai servizi informatici mediante interfacce utente molto ricche ed accattivanti usufruite attraverso browser;
- Service Oriented Architecture → permettono di organizzare le componenti computazionali distribuite, in modo da poterle identificare, allocare, orchestrare e ottimizzare per fornire funzionalità agli utenti;
- Autonomic computing → permette di arricchire i server e le funzionalità con la capacità di autogestirsi, in modo da limitare il bisogno di intervento umano nelle aree proprie della gestione.

12.14 Cloud computing (definizione)

Deriva dall'uso comune nel campo dell'Ing dell'Informazione di rappresentare Internet come una nuvola che tutto interconnette celando completamente l'infrastruttura estremamente complessa che tale servizio richiede. Vi è una generica convergenza sulla possibilità di definire 3 livelli di visione della nuvola:

- SW as a Service;
- Platform as a Service;
- Infrastructure as a Service.

12.15 Software as a Service (SaaS)

Si riferisce alla fornitura di un applicativo in modalità centralizzata e accessibile via Web. Una singola istanza dell'applicativo gestisce clienti diversi, pur garantendo la separazione logica dei dati di ciascun cliente.

I servizi applicativi offrono un'interfaccia via Web Services, che permette l'integrazione e l'interoperabilità con altri applicativi.

Cloud a questo livello consentono di sviluppare nuove applicazioni seguendo i principi SOA.

Tipici servizi che sono già resi disponibili al mercato, secondo la modalità SaaS, sono i servizi applicativi di base come back-up dei dati centralizzati, documentali, mail e archiviazione.

12.16 Platform as a Service (PaaS)

Il termine PaaS si riferisce alla modalità tramite cui una piattaforma rende disponibile, via Web, tutti quegli strumenti e prodotti utilizzati nello sviluppo e delivery di nuovi servizi applicativi.

I Cloud a questo livello offrono piattaforme che rendono disponibili strumenti di sviluppo come workflow di creazione di interfacce web, database integration, storage, integrazione di web-service. Tipi esempi sono:

- Azure Services Platform di MS;
- AWS;
- BlueCloud promossa da IBM;
- Google App Engine.

12.17 Infrastructure as a Service (IaaS)

Il termine IaaS si riferisce alla modalità di offrire come servizi infrastrutturali, risorse di elaborazione, memoria e comunicazione come:

- VM;
- CPU;
- Memoria;
- Schede LAN;
- Apparati di rete e loro configurazione;
- Servizi di backup.

I clienti accedono ai servizi tramite Internet, e il provider normalmente supporta il cliente con strumenti per la configurazione e il monitoraggio delle risorse.

13 Lezione del 26-04

13.1 Dispositivi palmari

Dotati delle capacità di collegarsi e sincronizzare dati i personal computer (seriale, USB, o BT). Si possono caricare programmi appositamente sviluppati. Alcuni modelli offrono anche connettività telefonica e di un hard-disk interno (tra i 2 e gli 8 GB).

13.2 Smartphone

Dispositivo portatile che abbina funzionalità di telefono cellulare a quelle di gestione di dati personali. La caratteristica più interessante degli Smartphone è la possibilità di installarvi ulteriori applicazioni, che aggiungono nuove funzionalità.

Il primo smartphone è Simon (progettato da IBM). La linea Communicator fu la prima della classe smartphone di Nokia iniziata nel 1996 con il modello Nokia 9000i.

13.2.1 I sistemi operativi

Un SO per dispositivi mobile deve rispettare vincoli in termini di dimensioni e limitatezza di risorse fisiche da un lato, e vincoli sulle prestazioni dall'altro.

Esaminando i primi SO per smartphone si evince come le soluzioni adottate siano state derivate da quelle dei tradizionali sistemi desktop ma al contempo come incomincino ad influire sulle nuove generazioni di tali sistemi.

Si arriva ad una progressiva riduzione del divario che ancora esiste fra le differenti tipologie di sistemi di elaborazione.

13.2.2 La memoria

La memoria viene, usata anche come spazio disco → necessario che il SO implementi tecniche molto sofisticate per l'uso della memoria disponibile.

13.2.3 L'alimentazione

L'alimentazione a batteria implica che un uso estensivo di risorse fisiche rischia di drenare rapidamente l'energia disponibile → il risparmio di energia impone una accurata gestione della CPU volta alla minimizzazione dei consumi elettrici.

13.2.4 Periferiche

Limitato numero di periferiche → si ricorre a schermi multi-touch.

Altra caratteristica importante è la robustezza del SO → si ricorre a tecniche di fault-tolerance (che risultano difficili da implementare).

13.2.5 Garanzie dei SO mobili

- Multitasking;
- Servizi soft real-time;
- Design architettonico.

13.3 Pattern Architetturali

13.3.1 Modello MVC (Model View e Controller)

13.4 Symbian

Approccio di tipo request and callback per i servizi, tiene separata l'interfaccia utente dal sistema, salvaguarda il consumo di energia ed implementa il pattern di progettazione orientato ad oggetti MVC. Nelle sue ultime versioni sono stati integrati dei moduli per la sicurezza e una versione real-time del kernel.

La programmazione in Symbian è basata su eventi e sulla tecnica active object. La gestione dei thread è realizzata in modo da non creare un eccessivo consumo di risorse.

13.4.1 Struttura del kernel

Presenta un microkernel che integra funzioni essenziali per garantire massima robustezza, sicurezza e reattività.

Layer

- Application Layer → comprende i componenti e l'interfaccia utente di ogni applicazione e usa i servizi messi a disposizione dai livelli di Middleware e Operating System;
- Middleware Layer → è suddiviso in domini applicativi (*es multimedia, networking etc*) che forniscono servizi al livello superiore;
- Operating System Layer → fornisce una serie di servizi di alto livello (comunicazione, networking, grafica, multimedia) e di basso livello (framework, librerie e utility);
- Adaptation Layer → integra la piattaforma SW generica con la piattaforma del telefono cellulare (Questo livello è implementato dal produttore del dispositivo);
- Cellular Platform → è il HW e il SW specifico del device che esegue i servizi richiesti dalla piattaforma Symbian.

13.4.2 Hardware

Symbian è stato progettato per garantire una elevata qualità dei servizi telefonici.

Convivono 2 componenti complementari:

- Mobile Radio interface del baseband (BP) o modem;
- Application Processor (AP)

Questi 2 processori hanno obiettivi molto diversi:

- BP → SW hard real-time
 - gestione periodica della carica;
 - sicurezza sulla rete;
- AP → framework e librerie per applicazioni native o di terze parti

13.4.3 MMU

Ha il compito di tradurre indirizzi virtuali in indirizzi fisici, così da rispondere ai primi 2 punti critici relativi alla gestione della memoria nei sistemi open. L'allocazione della memoria è gestita in 2 fasi:

- Allocazione di indirizzi virtuali (reserving);
- Allocazione di indirizzi fisici (committing);

Il sistema effettua il mapping fra indirizzi virtuali e fisici, attraverso un sistema multi-livello di page directory.

I page frame possono avere dimensioni di 4kb o 1Mb

13.4.4 Gestione della memoria

La page directory è costituita da 2^{12} voci, ciascuna di 4 byte, per una occupazione di memoria pari a 16Kb, L'indirizzo della page directory è conservato nel **Translation Table Base Register** (TTRB). L'indirizzo logico è diviso in 3 parti:

- 12 bit più significativi, utilizzati per indicizzare la **page directory**;
- 8 bit usati per indicizzare la **page table**;
- 12 bit utilizzati come **offset**.

13.4.5 I thread

Gestita dallo scheduler, per l'allocazione della CPU.

Costituito da un insieme di strutture dati, che descrivono, tra le altre cose, il punto del programma che esso ha raggiunto.

Un processo è un insieme di thread, che condividono lo stesso spazio di indirizzi.

Di conseguenza, ciascun thread può leggere o scrivere le zone di memoria di tutti thread, che appartengono allo stesso processo.

13.4.6 Lo scheduling

Gestito dal nanokernel attraverso un algoritmo di con priorità e prelazione, in cui il thread con priorità più alta viene eseguito.

I thread considerati eseguibili sono marcati come ready e sono accodati in una ready list.

Ciascun thread ha associata una priorità rappresentata con un valore intero fra 0 e 63. Se vi sono 2 thread con la stessa priorità, il modo con cui vengono schedulati, dipende da una proprietà impostata nel thread stesso:

- RR → la durata del time slice viene determinata sulla base delle proprietà del thread;
- FIFO → il primo thread ready in una lista di priorità viene eseguito, finché non si blocca o non deve cedere l'utilizzo della CPU.

13.5 Iphone e IOS

Presentavano una versione ottimizzata del sistema operativo Mac OS X privato di alcune componenti non necessarie.

Basato su un nucleo basato su un sistema Darwin, derivato a sua volta dalla fusione di FreeBSD e Mach. Risultano quindi essere sistemi della famiglia UNIX-like, ma a differenza di Android, solo il nucleo può essere definito Open Source.

13.5.1 Struttura IOS

- Cocoa Touch → strato che lavora al più ad alto livello
 - Si occupa della gestione e del riconoscimento del touch e del multi-touch dell'utente ed è in grado di interpretare, in maniera corretta, le gesture compiute dall'utente.
- Media → rappresenta lo strato che contiene tutte le funzionalità e le librerie per la gestione di video, audio, animazioni e altro;
- Core Services → Implementa le utility di sistema come
 - Networking;
 - Lista dei contatti;
 - Preferenze del sistema inserite dall'utente;
- Core OS → cuore del SO. In questo strato sono
 - Gestiti i file system;
 - Implementate le funzioni per la sicurezza del dispositivo;
 - Gestiti i certificati e altro.

14 Lezione del 30-04

14.1 Struttura di un sistema IOS

14.2 Mach

Nasce come progetto accademico → si base sull'idea di implementare un core minimale di tipo OO, in cui gli oggetti sono progettati ciascuno per un compito ben specifico:

- processi;
- thread;
- CPU;
- memoria virtuale.

A differenza di altri sistemi, in cui un oggetto può instanziare altri oggetti e utilizzarne i metodi, in IOS gli oggetti comunicano solo mediante **messaggi**

I messaggi sono accodati e gestiti da ciascun oggetto con una politica FIFO

Questo tipo di kernel ha dei compiti ben specifici:

- Gestione dei thread;
- Allocazione delle risorse a gruppi di thread (task);
- Allocazione e gestione della memoria virtuale;
- Allocazione della CPU per i diversi processi (scheduling).

La sua struttura OO, permette di essere fortemente multitasking. Ciascun oggetto può essere implementato ed eseguire su uno qualunque dei processori presenti nel sistema.

14.2.1 Messaggi

Vengono scambiati dagli oggetti, mediante l'invio su porte dedicate. Possono essere locali o remote, e tale scambio di informazioni è del tutto trasparente per l'implementatore di applicazioni. Presenta un *header*, un *body* ed eventualmente un *trailer*.

Possono diventare molto complessi (richiedono informazioni o strutture aggiuntive). Si attiva un bit detto MACH_MSGH_BITS_COMPLEX attivo nell'header, che di solito è seguito da un descrittore.

14.2.2 Invio e ricezione di messaggi

I messaggi vengono inviati e ricevuti mediante la stessa funzione `mach_msg()`, che prevede una implementazione per la modalità kernel e per la modalità utente.

Nella versione originale di Mach, la funzione copiava effettivamente i dati sfruttando il microkernel. Nelle versioni successive, poichè lo spazio degli indirizzi è unico, viene pasato solo il puntatore al buffer contenente i dati.

L'invio di messaggi da parte di un oggetto è implementato a livello del kernel dalle funzioni `mach_msg_overwrite_trap()` e `mach_msg_send()` (quest'ultima non visibile a livello utente). Il processo di invio di un messaggio prevede:

- Richiesta dell'IPC space;
- Richiesta dello spazio di memoria virtuale;
- Verifica dell'integrità del messaggio;
- Calcolo della dimensione del messaggio;
- Allocazione del messaggio;
- Copia del messaggio;
- Copia dei privilegi per l'invio del messaggio;
- Invio del messaggio.

La ricezione di messaggi da parte di un oggetto è implementato a livello del kernel dalle funzioni `mach_msg_overwrite_trap()` (user-mode) e `mach_msg_send()` (kernel-mode). Il processo di ricezione di un messaggio prevede:

- Richiesta dell'IPC space;
- Richiesta dello spazio di memoria virtuale;
- Richiesta della coda IPC;
- Recupero del messaggio dalla coda;
- Restituzione dei risultati dell'operazione.

14.2.3 Le porte

Sono interi a 32 bit. Può ricevere da molti sender, ma presenta un solo receiver.

Ciascun messaggio ricevuto da una porta viene accodato e mantenuto, finchè non è stato effettivamente gestito.

14.2.4 Privilegi delle porte

Sulle porte vengono gestite richiedendo un **handle** relativo alla porta dell'oggetto corrispondente. Ciascuna porta è caratterizzata da privilegi:

- SEND;
- RECEIVE;

- SEND_ONCE;
- PORT_SET;
- DEAD_NAME.

L'oggetto con i diritti MACH_PORT_RIGHT_RECEIVE, è considerato l'effettivo proprietario della porta.

Sia le porte, che i privilegi possono essere passati da un oggetto all'altro, nel sistema, attraverso lo scambio dei messaggi.

Le porte sono registrate in modo globale (demandato ad un port naming server, che in XNU è launchd, con processo PID 1)

Siccome tutti i processi sono figli di launchd, essi ereditano il registro delle porte, nel momento in cui questi vengono creati.

14.2.5 Primitive di sincronizzazione

Si basa sulla facoltà di precludere l'accesso da parte di altre entità del sistema, se queste devono essere gestite in modo esclusivo da una di esse. Sfrutta 2 layer:

- HW specific layer → insieme di istruzioni assembly che forniscono l'esclusione in modo atomico
- HW agnostic layer → delle API che rendono il meccanismo indipendente dall'HW sottostante

Match implementa di conseguenza una serie di oggetti di sincronizzazione:

- Mutex;
- Semafori;
- Spinlock;
- Lock sets.

La maggior parte di questi oggetti sono parte di un Lock Group Object, che è costituito da una lista concatenata di oggetti lock dello stesso tipo

14.2.6 Mutex

Mutual exclusive object → implementa una variabile binaria di stato, che indica se la risorsa sia disponibile o meno.

Hanno lo svantaggio di limitare ad un solo thread l'accesso alla risorsa.

14.2.7 Read/Write block

Distinzione fra accesso in lettura o scrittura di una risorsa. Come i mutex, sono idle-wait object (Se la risorsa è occupata causano la sospensione del processo).

In alternativa gli spinlock lasciano il processo in esecuzione, facendo polling sulla risorsa finchè non si libera.

Se la risorsa rimane occupata per breve tempo, il sistema evita inutili context switch.

Se la risorsa rimane occupata per molto o si creano cicli di attese, il sistema si trova in una condizione di deadlock.

14.2.8 Semafori

Sono una generalizzazione dei mutex → permettono di contare il numero di processi in attesa su una determinata risorsa. A differenza dei mutex sono disponibili anche in user mode.

14.2.9 L'oggetto host

Mach fornisce un'astrazione della macchina su cui esegue mediante un insieme di macchine primitive. È una collezione di porte speciali e gestioni di eccezioni.

Fornisce:

- Informazioni sulla macchina;
- Accesso ai sottosistemi;
- Gestori di eccezioni di default.

14.2.10 L'oggetto clock

Un parametro importante per le API relative al clock è il tipo di clock, che può essere:

- SYSTEM_CLOCK;
- CALENDAR_CLOCK.

Tutti i clock vengono creati dal `kernel_bootstrap`, che alloca una `alarm_zone` globale.

14.2.11 Processor object

Astrazione della CPU.

In un sistema multicore, ciascun processore è rappresentato da un processor object differente.

Questo oggetto è la principale entità coinvolta nelle operazioni di scheduling, in quanto al suo interno è implementata una runqueue.

I diversi processor object sono organizzati in **processor set** (o pset). Ciascun pset è mantenuto in una delle 2 code:

- `active_queue`;
- `idle_queue`.

14.3 IOS - scheduling

Il microkernel Mach fornisce diverse importanti caratteristiche relative alla gestione delle risorse interne al sistema, quali dispositivi HW, memoria virtuale, e soprattutto, la CPU.

Gli aspetti da tenere in considerazione sono:

- Scheduling → la descrizione di task e thread e delle API che essi offrono;
- Scheduling primitives → concetti di alto livello e algoritmi utilizzati;
- Asynchronous Software Traps (ASTs) → le eccezioni e gli interrupt utilizzati ai fini dello scheduling;
- Exception Handling → il modo in cui Mach gestisce l'HW;
- Scheduling Algorithms → descrizione del framework di scheduling.

14.4 Scheduling primitives

Al pari di altri SO Mach gestisce i thread, piuttosto che i processi.

I SO classici forniscono una visione top-down, in cui un processo è costituito da uno o più thread.

Al contrario Mach segue uno schema bottom-up, in cui un task è visto come un contenitore di thread.

14.5 IOS - I thread

Il thread costituisce l'unità di esecuzione in Mach.

Un thread è concepito per fornire la massima quantità di informazione necessaria allo scheduling, con il minimo overhead possibile.

La struttura che definisce il thread è molto complessa.

Il sistema crea una nuova struttura thread e la inizializza, clonando una struttura standard usata come template.

Un nuovo thread viene creato invocando la funzione `thread_create()`.

Durante il boot del sistema viene invocata la funzione `thread_bootstrap()`, la quale imposta i campi del template di thread, che fa da riferimento.

La funzione `thread_create()`, chiama la funzione `thread_create_internal()`, che clona il template di riferimento in un nuovo thread.

14.6 IOS - I task

Un task è un oggetto contenitore, all'interno del quale vengono gestiti la memoria virtuale ed il resto delle risorse.

Un task è quindi qualcosa di diverso da un processo, così come questo è definito negli altri sistemi operativi.

Tuttavia, MacOS e IOS si basano sullo schema BSD, che implementa il concetto di processo. Per tale ragione, per ciascun task in Mach esiste un processo in BSD, collegato attraverso il puntatore `bsd_info`.

Lo stesso kernel è implementato come task (noto come `kernel_task`) che non ha un PID (o in alternativa si potrebbe immaginare che abbia PID 0).

14.7 IOS - le API di Task e Thread

Le strutture di task e thread sono molti grandi e contengono molti dettagli e dati, che non sempre vengono gestite direttamente dalle API del kernel.

Due fra le funzioni più importanti per il kernel consistono nel determinare il thread corrente ed il task corrente.

Mach implementa una API `current_thread()`, la quale si riconduce alla macro `CPU_DATA_GET(cpu_active_thread, thread_t)` implementata con codice assembly inline.

Il task corrente viene recuperato leggendo il campo task del thread corrente.

14.8 IOS - Le priorità

A Ciascun thread è associata una priorità, che va da 0 (priorità bassa) a 128 (priorità alta). I processi in kernel mode hanno una priorità maggiore o uguale ad 80, mentre tutti i processi in user mode hanno priorità minore di 80.

In IOS le app in background hanno priorità pari a 4, mentre la app corrente ha priorità 47 ed il processo **SpringBoard** ha priorità 63.

La priorità assegnata inizialmente ai thread, viene riaggiustata durante l'esecuzione, a seconda dell'uso della CPU, attraverso:

- La macro `do_priority_computation();`
- La funzione `update_priority();`

14.9 IOS - Le run-queue

Al fine di rendere il processo di scheduling efficiente ISO si serve delle run queue, che vengono implementate con delle priority list.

Le run queue sono implementate con un array di 128 liste, una per ciascun livello di priorità. L'algoritmo di scheduling è analogo a quello utilizzato in Linux 2.6 con complessità O(1).

Lo scheduler aggiorna le priorità dei thread e li sposta da una run queue all'altra, in accordo alla priorità ricalcolata.

14.10 IOS - Le wait-queue

Una app può trovarsi in 5 stati differenti:

- Not running → non è in esecuzione o è stata terminata dal sistema;
- Inactive → esegue in foreground ma non riceve eventi;
- Active → esegue in foreground e riceve eventi;
- Background → lavora in background ed esegue codice;
- Suspended → lavora in background e non esegue codice.

Un thread che esegue una operazione bloccante, quale la richiesta per una risorsa, viene inserito in una wait queue, che vengono gestite da funzioni del kernel thread.

I thread vengono inseriti nelle wait queue mediante le funzioni `wait_queue_assert_wait()` (con diverse varianti).

I thread vengono sempre inseriti in coda alla lista. Vengono inseriti in testa se si tratta di:

- Un thread realtime;
- Un thread privilegioso;
- La coda è gestita in modalità FIFO.

Quando l'evento per il quale il thread attende si verifica, esso viene risvegliato mediante una delle funzioni:

- `wait_queue_wakeup64_one_locked();`
- `wait_queue_wakeup64_all_locked();`

IOS implementa il meccanismo della CPU affinity, con un meccanismo di `thread_binding`.

14.11 IOS - Lo scheduler

Mach affianca ai meccanismi di scheduling, comunemente usati anche da altri SO, alcune peculiarità:

- Handoff → permette ad un thread di specificare il thread che deve subentrare in un context switch;
- Continuation → permette di ignorare le informazioni dello stack durante un context switch;
- Selezione dello scheduler a tempo di boot.

14.11.1 Handoff

La gran parte dei SO prevede che un thread rilasci volontariamente la CPU. Tuttavia Mach, permette ad un thread di indicare il thread che deve subentrare ad esso.

Lo scheduler, tuttavia, non è obbligato a seguire tale direttiva,

In un context switch, seguito ad un handoff, il quanto di tempo del thread uscente è trasferito al thread entrante.

Il kernel implementa la funzione `thread_switch()`, che specifica la porta del thread entrante. La funzione è visibile anche in user mode, mediante la trap (#61).

14.11.2 Continuation

È una funzione opzionale di `resume`, che può essere specificata da un thread quando cede volontariamente l'uso della CPU.

Quando viene specificata una continuation, lo stack del thread non viene salvato e l'esecuzione del thread al successivo caricamento comincia con la prima istruzione di tale funzione, con uno stack completamente nuovo.

Un context switch con continuation richiede solo 4-5 Kb, invece dei circa 16 kb necessari in condizioni normali.

14.12 IOS - la prelazione

In IOS un thread è soggetto a prelazione in 2 modalità:

- Esplicitamente → cede il controllo della CPU o esegue una operazione bloccante;
- Implicitamente → il sistema riceva un interrupt.

La prelazione esplicita è considerata sincrona, in quanto è possibile prevedere a priori il verificarsi di tale condizione. Al contrario, quella implicita è considerata asincrona.

14.13 Le policy

La funzione `thread_policy_set()`, permette di impostare diverse policy:

- STANDARD_POLICY;
- EXTENDED_POLICY;
- TIME_CONSTRAINT_POLICY;
- PRECEDENCE_POLICY;
- AFFINITY_POLICY;
- BACKGROUND_POLICY;

La policy di default è THREAD_STANDARD_POLICY, che implementa il time sharing nella sua forma standard.

La policy THREAD_EXTENDED_POLICY prevede un parametro booleano, che se impostato a false indica l'utilizzo di una policy alternativa, e se true indica l'utilizzo della policy standard

La policy THREAD_AFFINITY_POLICY assicura che un thread possa eseguire sempre sulla stessa CPU. Un thread può eseguire sempre sulla stessa CPU.

Un thread può eseguire anche su core differenti della stessa CPU, ma non può passare da una CPU all'altra. Questo tipo di policy ottimizza l'utilizzo delle cache.

La policy THREAD_BACKGROUND_POLICY viene specificata per i thread in background, che hanno una priorità meno rilevante per il sistema come le app mandate in background dal modulo SpringBoard.

14.14 Internet of Things (IOT)

Coniata circa 20 anni.

L'IOT si è reso pervasivo nella nostra vita di tutti i giorni, fondamentalmente grazie a 2 fattori:

- Inanzitutto, il gran numero di servizi che permette di offrire lo ha reso quasi indispensabile;
- In secundis, grazie alla pletora di ambiti e campi di applicazione cui si può applicare.

Il tutto in maniera quasi trasparente all'utente finale.

14.14.1 Di cosa di tratta

Alla base dell'IoT vi sono degli smart object che hanno il compito di tastare l'ambiente ed acquisire informazioni tali da inferirne di nuove, prevedere situazioni e fornire servizi ad hoc per l'utente, per la collettività o per l'ambiente stesso.

Si intende quel percorso nello sviluppo tecnologico in base al quale attraverso la rete Internet, potenzialmente ogni oggetto dell'esperienza quotidiana acquisita una sua identità nel mondo digitale. L'Iot si basa sull'idea di oggetti "intelligenti" tra loro interconnessi in modo da scambiare le informazioni possedute, raccolte e/o elaborate.

La IoT va oltre gli oggetti intelligenti e assume un significato pieno nella rete che interconnette questi oggetti. Tra gli esempi ricordiamo:

- Automobili → box GPS-GPRS con finalità assicurative;
- Domotica → servizi cloud e altri servizi legati all'utilizzo crescente dell'AI;
- Industrie → ambito in cui le tecnologie IoT stanno contribuendo sia in termini di distribuzione dell'intelligenza del sistema che in termini di automazione;
- Città → pensiamo ai lampioni delle nostre città, in grado di regolare la loro luminosità sull'abase delle condizioni di visibilità.

14.14.2 Diffusione

Uno dei fattori che maggiormente ha permesso la sua diffusione è stato il concetto di Cloud → questo svolge la funzione di aggregatore, custode ed elaboratore dei dati generati dai processi e dagli Smart Object.

14.14.3 Ambiti Applicativi

- Smart Agriculture;
- Smart Car;
- Smart City;
- Smart Home;
- Smart Metering;
- Industrial IoT.

14.14.4 Sensori e Smart Objects

Nella maggior parte dei casi, uno smart object include uno o più sensori, a seconda del tipo di rilevazioni che devono essere eseguite.

In altri casi, un sensore può essere provvisto di un modulo esterno per fornire connettività al sensore stesso (trasformandolo così in uno smart object).

14.14.5 Discipline e attività di ricerca collegate alla IoT

- Elaborazione di Segnali, Immagini e Video;
- Riduzione dimensionalità
 - Attività necessaria in quanto spesso è necessario ridurre la quantità di dati ottenuti per facilitare le successive elaborazioni, riducendo a contempo la perdita di informazioni rilevanti;
- Scene Understanding

- Comprendere tramite segnali, immagini e video cosa sta accadendo in un particolare ambiente;
- Riconoscimento Biometrico
 - Riconoscere un individuo tramite le sue caratteristiche visibili e misurabili;
- Data Fusion
 - Siccome le sorgenti di dati, possono essere di tipi diversi, la data fusion si occupa di fondere tali dati in una rappresentazione unificata;
- Security
 - Data protection;
 - Biometric Authentication;
 - Cryptography;
- AI
 - Recommended Systems;
 - Feature Extraction;
 - Pattern Analysis.

15 Lezione del 03-05

15.1 Android OS

SO open source per dispositivi mobili, inizialmente basato su kernel 2.6 di Linux.

15.2 OHA

La Open Handset Alliance, con a capo Google, è una unione di:

- Produttori di dispositivi mobili
 - Motorola;
 - Sprint-NexTel;
 - HTC;
 - SamsungM
 - Sony;
 - Toshiba;
- Operatori telefonici
 - Vodafone;
 - T-Mobile;

- Costruttori di componenti
 - Intel;
 - Texas Instruments.

che ha come scopo quello di creare standard aperti per dispositivi mobili.

15.3 Nascita di Android

Nel novembre del 2007 la OHA viene istituita ufficialmente e presenta il SO Android, seguito dal rilascio del primo SDK per gli sviluppatori. Google presentò un intero ecosistema, un SO capace di funzionare su molti dispositivi diversi tra loro.

15.3.1 Le versioni di Android

Le versioni del SO sono indicate a livello ufficiale da un numero di versione seguito sempre da un nome in codice per tradizione ispirato a prodotti dolciari sempre in ordine alfabetico.

Android 1.0 e 1.1 → prime 2 versioni del SO rilasciate rispettivamente fine 2007 e inizio 2009 (con il primo update non vi furono novità significative)

Android 1.5 - Cupcake (aprile 2009) → primo major update:

- Introduce importanti novità a livello utente
 - Migliore esperienza con la tastiera;
 - Animazioni tra schermate;
 - Miglioramenti sugli aspetti multimediali.

Android 1.6 - Donut (settembre 2009) → Evoluzione della 1.5:

- Supporta reti CDMA (protocollo di accesso multiplo a canale condiviso di comunicazione, molto diffuso nelle reti wireless);
- Risoluzioni grafiche multiple.

Android 2.0, 2.1 - Eclair (ottobre 2009):

- Supporto multi account;
- Meccanismo di riconoscimento vocale.

Android 2.2 - Froyo (Maggio 2010) → Segna la scelta di produttori di entrare nel mercato delle piattaforme aziendali, oltre che quello consumer. Introduce:

- Compilazione JS just in time;
- Implementa il supporto al Flash Player;
- Il supporto al tethering USB e WiFi;
- Installazione di app su memoria esterna.

Android 2.3 - Gingerbread (dicembre 2010) → è servita soprattutto come porta di ingresso per il mercato dei giochi, ma anche per mettersi in pari con la concorrenza in qualche aspetto non del tutto secondario, come ad esempio la gestione del copia-incolla.

Android 3.0, 3.2 - Honeycomb (febbraio 2011e) → versione creata appositamente per dispositivi tablet con miglioramenti sia a livello di sistema sia specifici per il diverso ambiente di utilizzo.

Android 4.0 - Ice Cream Sandwich (Ottobre 2011) → rilasciata con l'intento di fondere tablet e smartphone. Viene presentato un nuovo SDK unificato e rende lo sviluppo di applicazioni per entrambi i segmenti di dispositivi molto più agevole.

Android 4.2 - Jelly Bean:

- Ottimizzazione dell'uso della CPU;
- Scrittura vocale offline;
- Ricerche intelligenti.

Android 4.4 - KitKat:

- Rinnovamento della UI e introdotto il full-screen completo;
- Hangouts diventa il client ufficiale per SMS e MMS;
- Diminuisce il consumo di batteria durante la riproduzione audio;
- Introduzione di ART (Android RunTime);
- Nuovo compilatore (sperimentale)

Android 5.0 - Lollipop:

- UI rinnovata;
- Eliminazione della runtime Dalvik in favore di ART;
- Miglioramento delle prestazioni grafiche grazie al supporto all'OpenGL ES 3.1;
- Aggiunta la modalità multi-utente.

Android 6.0 - Marshmallow (28 maggio 2015):

- Gestione personalizzata dei permessi per ogni singola applicazione;
- Collegamenti alle app più veloci;
- Nuovo sistema di risparmio energetico Doze;
- API per il supporto alle impronte digitali.

Android 7.0, 7.1 - Nougat:

- Nuova grafica nella barra delle notifiche;
- Nuovo menu di multitasking che elimina da sè le app inutilizzate;
- Supporto nativo al dual-window su smartphone e tablet.

Android Oreo (21 agosto 2017):

- Limiti per i processi in background delle app, in particolare per l'accesso alla posizione;
- Google toolchain;
- NTFS file system enabled

Android Pie (7 marzo 2018):

- Adaptive battery;
- Miglioramenti alla gestione del traffico in caso di rete congestionata;
- Accesso Multicamera.

15.4 Struttura del sistema

Strutturato come uno stack di SW, in cui sono inclusi:

- Il SO;
- Applicazioni middleware (intermediare fra app e componenti SW);
- Applicazioni chiave (fondamentali per il corretto funzionamento delle attività di base del sistema offerte all'utente finale).

15.5 Applicazioni

Qui si trovano i SW per l'interazione con l'utente. Il sistema fornisce un insieme di **core applications** per lo svolgimento delle funzionalità di base alle quali è possibile aggiungerne di nuove (via market o altri canali).

Ogni applicazione è scritta in Java.

15.5.1 Application framework

È una piattaforma per lo sviluppo e la creazione di nuove applicazioni da parte degli sviluppatori. La piattaforma dà anche pieno accesso alle API utilizzate dal sistema. Questa strutturazione favorisce il riutilizzo di ogni singolo componente.

Viene fornito anche un insieme di servizi:

- Un insieme di view per costruire le applicazioni;
- Content Providers, per comunicare fra applicazioni;
- Resources Manager, per l'accesso alle risorse grafiche;
- Notification Manager, per la gestione dei messaggi e delle notifiche a video;
- Activity Manager, per la gestione dei cicli di vita delle applicazioni.

15.5.2 Le librerie

Le librerie C/C++ sono utilizzate da diversi componenti del sistema. Le diverse funzionalità sono fornite agli sviluppatori attraverso l'Application Framework. Fra le librerie troviamo:

- Una libreria C che equivale all'implementazione standard libc ma adattata a sistemi Linux embedded;
- Librerie per la multimedialità;
- Librerie per la gestione della grafica;
- Una libreria dedicata per il Web Browser (LibWebCore);
- Una libreria per la gestione di database relazionali (SQLite).

15.5.3 Runtime

L'Android runtime è un insieme di librerie derivanti direttamente da Java e delle funzionalità legate all'avvio delle applicazioni.

La vecchia VM Dalvik → basata su tecnologia JIT (just-in-time); ogni app viene compilata in parte dallo sviluppatore e di volta in volta un interprete SW (la Dalvik) esegue il codice.

Il nuovo ART → basato su tecnologia AOT (ahead-of-time) che esegue l'intera compilazione del codice durante l'installazione dell'app e non durante l'esecuzione stessa del SW.

Quest'ultima fa riferimento al kernel sottostante per funzionalità di basso livello come la gestione di thread e della memoria.

15.5.4 Il Kernel

La piattaforma si basa sulla versione 2.6 del kernel Linux (3.x da Android 4.0 in poi) usato per i servizi di base del sistema:

- Sicurezza;
- Gestione della memoria;
- Gestione dei processi;
- Gestione della rete;
- Gestione dei driver.

Esso funziona anche come un livello di astrazione tra l'HW e il resto dello stack SW.

15.5.5 Applicazioni

Il funzionamento del SO ruota quasi completamente attorno alle applicazioni in esecuzione e alla loro interazione con l'utente.

Le applicazioni di sistema utilizzano le stesse API pubbliche, utilizzate dagli utenti.

Il SO offre delle applicazioni standard che possono essere sostituite su scelta dell'utente in caso di bisogno. Ogni applicazione lanciata ha associato un proprio processo con l'ID dell'utente.

Ogni applicazione ha la propria copia dell'ART, al fine di limitare il propagarsi di errori.

15.5.6 Gli intent

Le activity, i servizi e i broadcast receiver vengono tutti attivati da Intent. Questo è un messaggio asincrono che richiede l'avvio di una determinata azione. È possibile:

- Chiedere al sistema l'attivazione di un componente specifico;
- Chiedere al sistema l'attivazione di un tipo componente (è compito del sistema associare alla richiesta un componente disponibile in grado di soddisfarla).

Un oggetto Intent è un aggregato di informazioni:

- Informazioni di interesse per il componente ricevente (*es.l'azione da effettuare*);
- Informazioni di interesse per il Sistema Android.

Gli intent possono essere suddivisi in 2 gruppi:

- Espliciti → indirizzati direttamente verso un componente identificato dal proprio nome
 - Vengono utilizzati solitamente in modo trasparente all'utente per le comunicazioni interne ad un'applicazione;
- Impliciti → quelli per i quali non viene specificato un componente (il campo nome resta vuoto)

- Vengono spesso utilizzati per attivare componenti di altre applicazioni.

Un intent è formato da:

- Nome del componente a cui è indirizzato (campo opzionale);
- Azione che deve essere svolta (identificata da una stringa);
- URI dei dati e tipo di dati su cui deve essere svolta l'azione;
- Categoria, ovvero una stringa con informazioni legate al tipo di componente a cui è indirizzato l'intent;
- Un campo extra per l'inserimento di informazioni aggiuntive;
- Un campo flag.

15.5.7 Le Activity

Una Activity è un componente delle applicazioni, che fornisce un'interfaccia tramite la quale l'utente può interagire.

Una applicazione è costituita da più activity, di cui una è specificata come principale e viene visualizzata sullo schermo appena l'utente lancia l'applicazione per la prima volta.

Ogni classe implementata deve o creare sul momento gli elementi visuali su cui poggia o deve aver associato un file .xml nel quale sono dichiarati gli elementi grafici che saranno visualizzati.

Ogni activity è caratterizzata da un ciclo di vita ed in ogni istante si trova in uno dei seguenti stati:

- **Active/Resumed** (running) → essa è visibile e può ricevere input dall'utente;
- Paused → essa è parzialmente visibile a causa di un'altra activity in foreground, non può ricevere input dall'utente (*es* è in attesa a causa di un alert dialog presente al di sopra di essa);
- Stopped → essa non è visibile (resta in background)

Si possono individuare 3 diversi sotto-cicli all'interno del ciclo di vita di una activity:

- L'intero tempo di vita, compreso fra il lancio di `onCreate()` (dove avviene il setup dello stato globale e dell'ambiente) e di `onDestroy()` (dove vengono rilasciate le risorse utilizzate).
- Il tempo di vita nel quale l'activity è visibile, compreso fra i metodi `onStart()` e `onStop()`. In questo frangente l'activity è visualizzata a schermo dall'utente, il quale potrà interagire con essa;
- Il tempo di vita nel quale l'activity è in foreground, compreso fra i metodi `onResume()` e `onPause()`.

In Android viene associata un'activity ad ogni orientazione (portrait e landscape). Ogni cambiamento fra una e l'altra causa una transizione degli stati paused, stopped e destroyed per quella attività, mentre viene istanziata quella nuova.

Lo stato di un'activity è l'insieme delle informazioni degli elementi di cui è composta in un dato istante.

Quando un'activity ritorna in foreground, il suo stato non è garantito che permanga nel tempo.

È necessario implementare un ulteriore metodo (`onSaveInstanceState()`) per essere sicuri che venga correttamente ripristinata

15.5.8 I Services

È un componente di un'applicazione che si occupa di svolgere in background operazioni che richiedono **lunghi tempi** di esecuzione. Per sua natura non è provvisto di interfaccia utente.

Una qualsiasi applicazione può lanciare un servizio che continuerà a rimanere in esecuzione anche nel momento in cui l'utente deciderà di eseguirne un'altra.

Un qualsiasi componente può legarsi ad un servizio per interagirvi ed eventualmente comunicare con altri processi (IPC).

Può assumere 2 forme:

- **Unbounded**

- Un componente dell'applicazione lo lancia attraverso `startService()`;
- Può rimanere in background per un tempo indefinito, anche nel momento in cui chi l'ha lanciato viene distrutto;
- Le funzioni di servizi di questo tipo sono singole, ovvero ogni servizio svolge una singola operazione e non restituisce alcun valore al chiamante;
- Ad operazione conclusa è lo stesso servizio a fermarsi.

- **Bounded**

- Un componente lo lega a se stesso attraverso `bind-Service()`;
- Offre un'interfaccia client-server che permette al componente di interagire (es. invio di richieste, comunicazione con altri processi);
- Il tempo di vita del servizio è strettamente legato alla vita del componente che l'ha lanciato; alla distruzione del componente viene distrutto anche il servizio;
- Allo stesso servizio possono inoltre legarsi più componenti e in questo caso continuerà a rimanere in background fin tanto che almeno uno vi è ancora collegato.

Una volta avviato, un servizio può essere utilizzato da un qualsiasi componente dell'applicazione (o da altre applicazioni), a prescindere da chi lo abbia avviato.

È possibile dichiarare un servizio di tipo privato, bloccando così l'accesso da parte di altre applicazioni.

Un servizio avviato è legato al thread principale del processo dell'applicazione, non crea (se non specificato) un proprio thread separato e non viene avviato in un altro processo.

15.5.9 Content Provider

Rappresentano l'interfaccia standard per la condivisione di dati fra più processi.

Essi regolano l'accesso a insiemi di dati strutturati, occupandosi dell'incapsulamento e della sicurezza degli stessi.

L'accesso ai dati gestiti da un content provider avviene inoltrando una richiesta ad un oggetto di tipo **ContentResolver** all'interno del contesto dell'applicazione stessa.

Il SO mette a disposizione un set di provider di base. I dati sono organizzati in tabelle ed un content provider può gestire più tabelle. Per ogni tabella è presente una colonna ID (chiave primaria), la quale associa ad ogni record un numero unico identificativo.

Un'applicazione che non necessita di condividere i propri dati con altre non ha bisogno di implementare un proprio content provider, a meno che non debba effettuare trasferimenti di dati complessi o file.

Per identificare i dati si utilizzano stringhe basate sullo standard URI (Uniform Resource Identifier), in cui ciascuna stringa segue uno schema fisso per identificare delle risorse.

Ogni stringa URI segue il seguente schema:

- Nome dello schema (content nel caso si utilizzano content provider);
- ‘://’;
- Autorità (in questo caso il nome del content provider);
- Path al quale è localizzata la risorsa.

15.5.10 Broadcast Receiver

È un componente che si attiva rispondendo ad un messaggio inviato in broadcast dal sistema stesso.

Il ciclo di vita di un broadcast receiver può essere riassunto nei seguenti punti:

- Il receiver viene registrato;
- Quando viene mandato un intent in broadcast che può attivare il ricevitore, viene richiamato il metodo `onReceive()`;
- Quando `onReceive()` si conclude, l'oggetto ricevitore non è più attivo.

Android impone al metodo `onReceive()` di completare la sua esecuzione entro 10 secondi. Allo scadere del timer il ricevitore viene considerato bloccato e il processo può essere cancellato.

15.6 Processi e Thread

Si definisce processo un'istanza di un programma attualmente avviato dal sistema operativo. Processi diversi non condividono risorse.

Si definisce thread un'unità di un processo gestita dallo scheduler del sistema operativo.

Un task può contenere più thread che condividono le stesse risorse.

Il multitasking utilizzato da Android è di tipo **preemptive**, ovvero i thread vengono rimossi forzatamente dall'unità di esecuzione quando lo decide il SO.

15.6.1 Multitasking

In Android non possono essere eseguite più applicazioni visibili a schermo pieno contemporaneamente. Tuttavia è possibile gestire più thread concorrentemente ed è stato introdotto il multi-view.

Le activity in background continuano ad essere schedulate: tuttavia, in qualsiasi momento possono essere fermate o distrutte da Android senza alcun preavviso, per cui non vi è la cercetta che un'activity in tale stato completi sempre il suo lavoro.

In Android il multitasking è strettamente legato al ciclo di vita delle activity e alle operazioni che compie l'utente.

15.6.2 Stack delle activity

In Android ciascuna activity può invocarne altre, che vengono mantenute in uno stesso stack.

Si definisce **task** (per Android) una collezione di activity con le quali l'utente interagisce quando deve compiere una determinata operazione.

Quando l'utente lancia un'applicazione, il task corrispondente viene messo in foreground. Nel caso non ne esistano per l'applicazione appena avviata, viene creato un nuovo task e l'activity principale dell'applicazione viene inserita come radice in uno stack, chiamato **back stack**.

Ogni qual volta un'activity viene lanciata viene subito inserita in cima a tale stack dal sistema. La precedente activity è preservata nello stack in stato di stop.

Un task, essendo un insieme di più activity, viene messo in background assieme all'intero stack, quando l'utente inizia un nuovo task o ritorna all'Home screen attraverso l'Home button.

Dato che in uno stack le posizioni non vengono mai riordinate, se un'applicazione lo permette è possibile avere nello stesso stack più istanze della stessa activity.

Se un componente di un'applicazione viene lanciato ed esiste già un processo per tale applicazione esso viene eseguito nello stesso processo e utilizza lo stesso thread di esecuzione.

15.7 Lo scheduling dei processi

Fino alla versione precedente la 2.6.23, il kernel vanilla aveva uno scheduler detto anche **O(1) scheduler**, che era in grado di eseguire i vari task in tempo costante e indipendentemente dal numero di processi presenti nella coda.

Uno scheduler del genere è ottimo per macchine di tipo server, ma per macchine che richiedono un buon grado di interattività non è propriamente adatto.

Per risolvere questo problema sono state introdotte una serie di patch rivolte all'interattività. Successivamente è stato sviluppato il Completely Fair Scheduler.

15.7.1 CFS

Non organizza i processi in code, ma in un albero binario del tipo red-black tree.

Quando un task entra nella coda, il suo time viene annotato e, via via che esso aspetta il momento di andare in esecuzione, è aggiornato dal kernel in base alla priorità del task.

Processi con maggior bisogno di utilizzare la CPU sono mantenuti nella parte sinistra dell'albero, viceversa processi con meno bisogno sono mantenuti nella parte destra.

Lo scheduler, seguendo il principio di equità, prende di volta in volta il processo più a sinistra.

15.8 La gestione della memoria

La gestione della memoria basata sulla paginazione:

- Un segmento di memoria consiste in una o più pagine;
- Indirizzi utilizzati dai processi sono virtuali, vengono tradotti in indirizzi fisici mediante paginazione;
- Indirizzi virtuali diversi possono essere tradotti nello stesso indirizzo fisico.

Rispetto al kernel 2.4 attualmente è supportato il meccanismo di **reverse mapping** che consiste nel mantenere per ogni indirizzo fisico una lista concatenata di puntatori alle Page Table Entries (PTEs) di ogni processo che attualmente sta mappando quella pagina.

Questo sistema consente di aggiornare velocemente le page table dei processi che mappano pagine che sono state spostate nell'area di swap.

15.9 Il power management

Linux implementa meccanismi di power management sui dispositivi laptop al fine di gestire le condizioni di sospensione e ibernazione.

Su un dispositivo mobile, le strategie di power management sono molto diverse.

Opportunistic suspend

Secondo la strategia di power management di Android, il normale stato del sistema è uno sleep state, in cui l'energia è utilizzata per il refresh della memoria e per i componenti in grado di generare un segnale di wakeup di sistema.

Il sistema rimane attivo per il solo lasso di tempo necessario ad eseguire un task.

Il primo step del **processo di sospensione** del sistema, consiste nel congelamento dei processi in user mode, i quali non potranno più catturare segnali inviati dal kernel.

Se un segnale arriva immediatamente dopo l'avvio del processo di sospensione, esso rimane in coda.

All'arrivo di un nuovo segnale di wakeup, il processo riceve quello precedentemente rimasto in coda.

L'evento di wakeup, rimasto in attesa poteva essere una chiamata (di estrema importanza) in ingresso, e che viene persa

15.10 Il wakelock

Un wakelock è un oggetto che presenta 2 stati:

- Attivo;
- Inattivo.

Il sistema non può essere sospeso, se esiste almeno un wakelock, che sia nello stato attivo.

Il kernel attiva un wakelock, subito dopo l'invio di un segnale di wakeup e lo disattiva subito dopo che questo è stato passato allo user space.

L'utilizzo di questi oggetti è particolarmente controverso, poichè anche i processi in user mode possono modificare lo stato, con conseguenze sul comportamento del sistema e sul consumo di batteria.

L' **opportunistic suspend** crea problemi a tutte quelle applicazioni, che necessitano di fare un **check periodico** di risorse o informazioni. Il check viene fatto quando il sistema è in wakeup e non quando realmente dovrebbe.

16 Lezione del 07-05

16.1 Problemi legati al timekeeping

Un grave problema legato alla sospensione, indotta sia in kernel mode che in user mode è rappresentato dall'aggiornamento dei timer.

Quando il sistema va in sospensione, l'HW relativo al timer è soggetto a power off e le variabili globali del kernel relative al tempo vengono aggiornate sulla base di un clock persistente, che è meno accurato.

Ad ogni ciclo **suspend-resume** queste variabili devono essere riallineate.

Quando l'operazione di sospensione è richiesta in user mode, il kernel assume che i processi in user mode, siano in grado di gestire tale fenomeno.

Il kernel, può prevedere di reimpostare i valori correnti da un (Network Time Protocol - NTP) server.

Al contrario, quando la sospensione è richiesta in kernel mode, i processi in user mode non possono predisporre tale aggiornamento, non prevedendo l'imminente sospensione.

Un alternativa consiste nel porre il sistema in uno stato di **cpuidle**, che è meno restrittiva della sospensione. Questo stato prevede che un certo numero di I/O device possano richiedere il wakeup del sistema.

Tuttavia, **cpuidle** tende ad abilitare tutte le I/O device autorizzate a causare il wakeup del sistema, riducendo i tempi effettivi di sleep e fornendo un risparmio energetico minore rispetto a quello garantito dall'**opportunistic suspend**.

Comportamenti incosistente del processo di sospensione rispetto agli eventi di wakeup, sono un problema per tutti i meccanismi di sospensione e non solo per l'opportunistic suspend.

La gestione dei processi in user mode in Android è stata progettata in modo da tenere conto dei meccanismi di wakelock; tuttavia non è detto che altri sistemi Linux based facciano lo stesso.

16.2 Wakeup events framework

Al fine di risolvere tali problemi in modo più generale, in Linux 2.6.36 è stata prevista una patch, che introduce il **wakeup events framework**. Questa introduce:

- Un contatore degli eventi di wakeup generati (`event_count`);
- Un contatore degli eventi in corso di elaborazione (`events_in_progress`);
- Un metodo che evita la sospensione del sistema (`pm_stay_awake()`);
- Un metodo che evita la sospensione durante un evento (`pm_wakeup_event()`);

Il metodo `pm_stay_awake()` incrementa la variabile `events_in_progress`.

Il metodo `pm_relax()` decrementa tale variabile e incrementa `event_count` successivamente.

Il metodo `pm_wakeup_event()` incrementa la variabile `events_in_progress` e imposta un timer, allo scadere del quale. ri-decrementa tale variabile e incrementa `event_count`.

16.3 Wake source object

Il limite di questo meccanismo è nel fatto che esso presuppone che gli eventi di wakeup siano sempre associati ad un device.

La soluzione è stata fornita in una patch successiva, che introduce i **wakeup source object**, implementati dalla struttura `struct wakeup_source`.

Questi oggetti vengono creati automaticamente per i device abilitati a inviare i segnali di wakeup e sono gestiti internamente dalle funzioni `pm_wakeup_event()`, `pm_stay_awake()`, e `pm_relax()`.

Essi possono essere gestiti attraverso funzioni specifiche dedicate:

- `wakeup_source_create()`;
- `wakeup_source_add()`;
- `__pm_wakeup_event()`;
- `__pm_stay_awake()`;
- `__pm_relax()`;
- `wakeup_source_remove()`;
- `wakeup_source_destroy()`;

Questo approccio permette l'implementazione di un power management system molto simile all'opportunistic suspend, semplicemente sostituendo i wakelock con dei wakeup source object.

I processi in user mode non possono accedere ai wakeup source object, per cui è necessario un meccanismo specifico per questo contesto, implementato completamente in user mode.

Questo meccanismo prevede l'utilizzo di 3 componenti:

- Una locazione di memoria condivisa contenente la variabile **suspend counter**;
- Un **mutex**;
- Una variabile di stato associata al mutex;

Un processo che vuole evitare la sospensione del sistema:

- Acquisisce il **mutex**;
- Incrementa il **suspend counter**;
- Rilascia il **mutex**.

Un processo che vuole permettere la sospensione del sistema:

- Acquisisce il **mutex**;
- Decrementa il **suspend counter**;
- Se suspend counter è 0, i processi in attesa su di essa vengono **sbloccati**;
- Libera il **mutex**.

Questo approccio implementa una **sospensione** (molto) **aggressiva** del sistema

16.4 La sicurezza in Android

Quando si parla di sicurezza, bisogna prendere in considerazione l'ambiente esterno, in modo da proteggere il sistema da:

- Accessi non autorizzati;
- Modifica o cancellazione di dati fraudolenta, perdita di dati o introduzione di inconsistenze "accidentali";

Gli smartphone presentano caratteristiche diverse rispetto ai computer tradizionali.

Non tutte le considerazioni tradizionali valide sono immediatamente applicabili.

16.4.1 Vulnerabilità - Accesso Fisico

Furto, Smarrimento o Incuria.

Esistono contromisure più o meno efficaci per limitare l'accesso fisico ad uno smartphone

16.4.2 System Partition and Safe Mode

La partizione di sistema è impostata come read-only e include:

- Il kernel;
- Le librerie di sistema;
- L'ART;
- Il framework delle applicazioni;

- Le applicazioni di sistema.

Il boot del sistema avviene in safe mode, in cui sono disponibili solo le Android core applications, assicurando che l'avvio avvenga in un ambiente privo di applicazioni di terze parti.

Android eredita il meccanismo dei permessi, secondo cui ogni processo può accedere solo alle proprie risorse.

Sebbene il programmatore esponga i propri file alle altre applicazioni queste ultime non possono accedervi, dato che eseguono in sandbox distinte.

16.4.3 Sicurezza a livello kernel

A livello del SO la piattaforma Android offre meccanismi di protezione:

- A livello kernel;
- A livello IPC.

I meccanismi di sicurezza assicurano che anche il codice di sistema venga eseguito in una **sandbox**, al fine di precludere attacchi da parte di applicazioni esterne.

Android eredita alcune di queste caratteristiche da Linux, da lungo tempo utilizzato in contesti, in cui la sicurezza rappresenta una criticità:

- Un modello di permessi user-based;
- Isolamento dei processi;
- Meccanismi sicuri di comunicazione interprocesso (IPC);
- Meccanismi per l'**eliminazione/sostituzione** di sezioni del kernel ritenute vulnerabili.

16.4.4 Modello di sicurezza in Android

Il modello di sicurezza proposto da Android è in gran parte ereditato dal modello di sicurezza del SO **Gnu/Linux**, tuttavia è stata fatta qualche modifica per adattarlo alle esigenze dei dispositivi mobili.

In Linux i processi sono tutti identificati da un numero intero chiamato User ID (**UID**) e quindi sono logicamente separati gli uni dagli altri.

Android isola i processi e le risorse come Linux ma assegna gli identificativi differentemente. Poichè non c'è bisogno di assegnare più di un UID per l'utente fisico, gli UID sono assegnati alle applicazioni.

16.4.5 Protezione a livello di applicazioni

A livello di applicazione, il sistema adotta diverse politiche per la protezione da componenti malevoli:

- Voluta assenza di API per funzionalità critiche (gestione della SIM);
- Separazione dei ruoli per le applicazioni;
- Thrusted application mediante Permissione per funzionalità quali
 - Camera;
 - GPS;
 - Bluetooth;
 - Telefonia;
 - SMS/MMS;
 - Network.

Le applicazioni devono dichiarare i diritti di cui necessitano nel file manifest.

Android segnala le cost sensitive API:

- Telefonia;
- SMS/MMS;
- Network/Data.

16.4.6 Le Sandbox

Ogni applicazione in Android è associata ad un ID (UID) ed esegue in un processo separato dal resto (Sandbox).

La Sandbox è implementata nel kernel e ciò comporta diversi vantaggi:

- È un meccanismo di protezione relativo sia al kernel, che al codice nativo;
- L'incremento del livello di sicurezza delle applicazioni non è carico del programmatore;
- Problemi di memory corruption non pregiudicano il funzionamento dell'intero sistema.

Il meccanismo delle Sandbox non è invulnerabile. Tuttavia, raggiungere questo livello di sicurezza significa riuscire a corrompere il kernel.

Il sandboxing è stato progettato con una duplice intenzione:

- Evitare che le applicazioni consumino tutte le risorse del SO;
- Evitare che le stesse applicazioni abbiano piena e incontrollata libertà di interagire tra loro.

All'atto dell'installazione di una app, Android gli assegna un UID univoco.

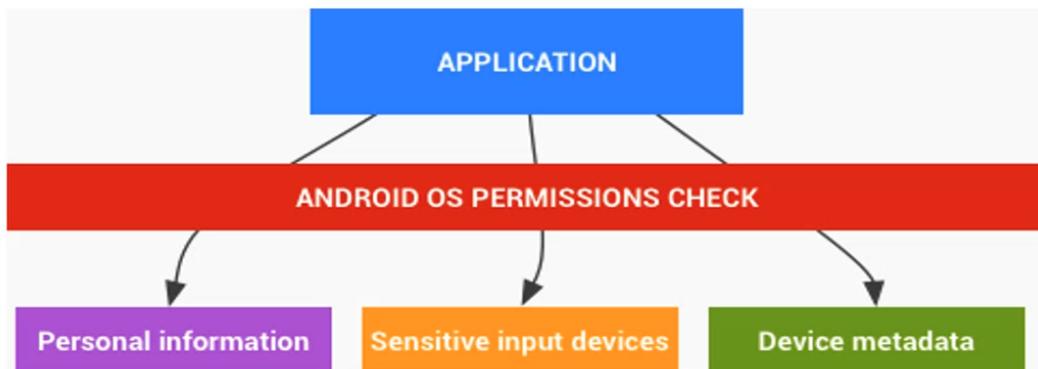
L'esecuzione dell'applicazione avviene poi all'interno di un processo che gira con il medesimo UID.

16.4.7 Protezione a livello di applicazioni

Le API che consentono l'accesso ai dati dell'utente sono incluse fra quelle che richiedono il check dei permessi.

Un'applicazione che desidera accedere ai dati dell'utente deve ricorrere alle politiche di verifica dei permessi di Android.

Android si basa anche sul meccanismo dell'application signing



16.4.8 Lo User ID

Gli UID sono generati ed assegnati seguendo un criterio che rispecchia anche i privilegi concessi all'applicazione o al system service:

- System service con UID 0 che ha privilegi da (**ROOT USER**);
- System service con UID che partono da 1000 (**AID_SYSTEM**) che hanno privilegi limitati;
- Normali app con UID che partono da 10000 (**AID_APP**).

Ogni app ha un username simbolico scritto nella forma `app_XXX`, dove le 3 x rappresentano lo scostamento dell'UID dell'app da quello dell' **AID_APP** (es. `app_37` ha UID $\text{AID_APP} + 37 = 10037$)

16.4.9 Lo Shared User ID

Android fornisce la possibilità alle applicazioni che sono strettamente accoppiate, cioè quelle che hanno bisogno di comunicare di frequente e di accedere alle stesse risorse, di essere installate con lo stesso UID.

Questo UID prende il nome di **shared user ID** e permette alle applicazioni di essere eseguite all'interno dello stesso processo.

I presupposti per poter utilizzare lo shared user ID sono 2:

- Le app richiedenti devono essere firmate con la stessa chiave;
- Le app devono richiedere lo shared user ID a tempo di installazione.

Molte app di sistema utilizzano lo shared user ID mentre questo comportamento per le app comuni è raro e sconsigliato.

16.5 I permessi

In Linux ad ogni file è associata una tripla di permessi:

- read (*r*);
- write (*w*);
- execution (*x*).

Le operazioni che possono essere richieste ad una cartella sono le stesse ma il significato è differente da quello dei file normali:

- Vedere la lista dei file della cartella (*r*);
- Aggiungere o eliminare file dalla cartella (*w*);
- Accedere a tutti i file della cartella (*x*).

Possiamo quindi suddividere gli utenti in 2 macrocategorie:

- Gli utenti non privilegiati (UID diverso da 0) o **normali**;
- Gli utenti privilegiati o **root** (UID = 0).

I permessi si dividono in 2 categorie:

- Preinstallati;
- Personalizzati.

Tutti i permessi sono definiti all'interno del package android e sono scritti nella seguente forma:

```
permission: android.permission.PERMISSION_NAME
```

Un'app richiede al SO uno o più permessi attraverso un file chiamato *AndroidManifest.xml* usando il tag `<uses-permission>`.

Android non pone i permessi tutti sullo stesso piano infatti esistono 4 livelli di protezione ed ogni permesso appartiene ad un solo livello.

Normal → il livello di protezione di base;

Dangerous → possono costituire un pericolo (es. gestione dei dati utente, controllo di risorse HW).

Signature → livello di protezione esclusivo, sono rilasciati solo alle applicazioni firmate con la stessa chiave crittografica dell'applicazione che ha dichiarato il permesso.

SingatureOrSystem → rilasciati ad app firmate dell'app o che fanno parte dell'immagine con la stessa chiave crittografica del possessore del sistema

Dopo che l'applicazione ha ricevuto tutti i permessi richiesti il sistema memorizza quali permessi sono stati rilasciati per quell'app utilizzando il concetto di **gruppo** in modo analogo a quello implementato da Linux.

Durante l'installazione l'app riceve un UID e viene eseguita in un processo che avrà lo stesso UID, un GID e diversi GID supplementari.

Questo meccanismo consente di restringere l'accesso alle risorse HW o di SO solo a chi appartiene ad un determinato gruppo.

Ad esempio tutte le app che hanno ricevuto il permesso per utilizzare connessioni internet appariranno al gruppo **inet**.

16.6 Meccanismi di controllo

Lo strato di **Application Framework** si occupa di controllare che un processo chiamante abbia tutti i permessi necessari per poter utilizzare le risorse che sta chiedendo.

Il **controllo dinamico** sfrutta il fatto che ogni UID è associato ad un solo package ed il **system service package** manager gestisce un database aggiornato di tutti i permessi rilasciati ad ogni package.

Il **controllo statico** ha luogo quando un'app cerca di utilizzare un metodo o una risorsa di un'altra e per far ciò l'app chiamante deve aver ottenuto un permesso specifico.

Android implementa questo meccanismo mediante gli **intent**

Dalla versione 6.0 di Android, non è più necessario rilasciare i permessi a tempo di installazione, infatti le app sono installate automaticamente ed i permessi necessari sono accordati a tempo di esecuzione.

16.7 La crittografia

Android fornisce un insieme di API per la crittografia alle applicazioni.

Queste API includono un insieme di primitive standard:

- AES;
- RSA;
- DSA;
- SHA.

Ai quali si aggiungono SSL e HTTPS.

A questi protocolli Android aggiunge la classe KeyChain, che permette di accedere allo storage di sistema delle credenziali per la gestione di chiavi e certificati:

- Viene inoltrata una richiesta di una nuova chiave al key manager X509KeyManager;
- L'utente seleziona una chiave ed il corrispondente certificato da una lista;
- Restituisce le credenziali selezionate alla callback ricevuta.

`createInstallIntent()` permette ad una applicazione l'installazione di chiavi e certificati.

16.8 Protezione del filesystem

A partire dalla versione di Android 3.0 il sistema fornisce la possibilità di criptare l'intero filesystem.

I dati utente possono essere criptati nel kernel sulla base di protocolli AES128.

La chiave utilizzata è a sua volta protetta con AES128 utilizzando una chiave derivata dalla chiave di accesso utente.

Al fine di proteggersi da attacchi brute force, la chiave è combinata con un seme random e soggetta a più livelli di hashing con SHA1.

Per garantire la protezione contro attacchi basati su dizionari, Adroid impone delle regole sulla complessità della password utente.

Nel caso il filesystem venga criptato, l'accesso al dispositivo con sblocco è disabilitato.

17 Lezione del 10-05

17.1 Le biometrie

Classificazione

- Cooperativo/non cooperativo → l'utente collabora all'acquisizione oppure no;

- **Evidente/Velato** → il sistema è nascosto al pubblico o meno;
- **Abituato/Non abituato** → il sistema è usato spesso dallo stesso utente o meno;
- **Frequentato/Non frequentato** → il sistema è usato più volte nell'arco della giornata anche da utenti diversi;
- **Pubblico/privato** → il sistema è accessibile ad un numero elevato di utenti diversi;
- **Aperto/Chiuso** → il sistema ha o non ha scambio di dati con l'esterno.

17.2 Qualificazione delle Biometrie

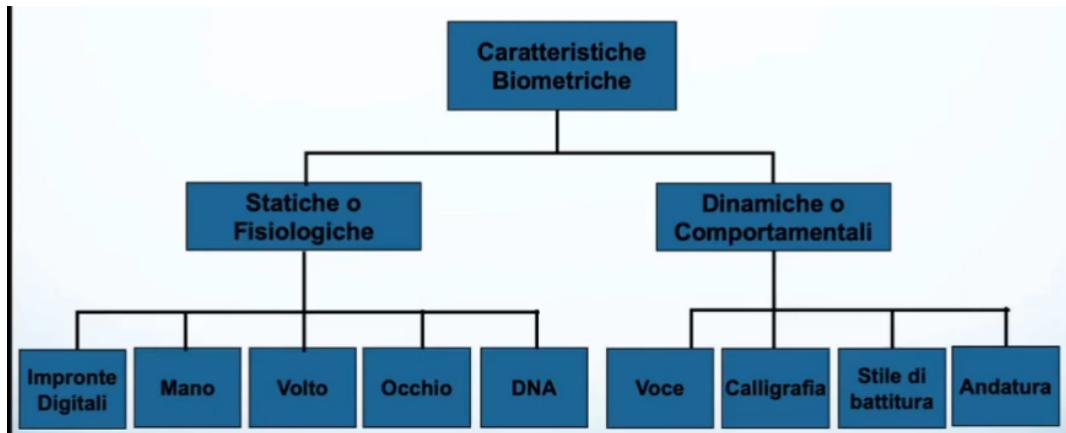
- Efficienza → costo computazionale in termini di **tempo/memoria**;
- Esecuzione (efficacia) → raggiungimento preciso dell'identificazione;
- Campi di applicazione → verifica o riconoscimento;
- Acquisizione → **online/offline**;
- Costi HW;
- Metodologia;
- Universalità → ogni individuo possiede o meno una determinata caratteristica;
- Unicità → il grado con cui si può trovare la stessa caratteristica tra 2 soggetti diversi;
- Permanenza (Stabilità) → caratteristica varia o meno nel tempo;
- Misurabilità → le caratteristiche possono essere misurate quantitativamente;
- Accettabilità → il grado con cui una persona è disposta ad utilizzare un determinato sistema biometrico;
- Insidia → il grado di poter ingannare il sistema utilizzando una determinata caratteristica biologica.

17.3 Comparazione di tecnologie biometriche

	Universalità	Unique	Permanenza	Collectability	Performance	Accettabilità	Unspoofability
Fingerprint	M	H	H	M	H	M	M
Iris	H	H	H	M	H	L	H
Retina	H	H	M	L	H	L	H
HandGeometry	M	M	M	H	M	M	M
PalmPrint	M	H	H	M	H	M	M
Hand Vein	M	M	M	M	M	M	H
Voice	M	L	L	M	L	H	L
Face	H	L	M	H	L	H	L
Face Therm	H	H	L	H	M	H	H
DNA	H	H	H	L	H	L	L

(H → hight; M → medium; L → low)

17.4 Caratteristiche Biometriche



17.5 Sicurezza dei Sistemi biometrici

Test di violabilità di sistemi biometrici in commercio hanno dimostrato che molti sono vulnerabili a:

- Digital spoofing
 - Attacchi replay → un hacker ruba l'immagine digitalizzata e se ne serve in una altra occasione;
 - Attacchi di manipolazione → del valore di soglia tipico di ciascun sistema (obiettivo → aumentare il FAR);
 - Inserimento nel sistema di un "Cavallo di Troia" per fornire dati erronei al programma di estrazione dei parametri biometrici dall'immagine scansionata;
 - Alterazion del risultato finale del processo biometrico → l'inserimento e l'analisi dei dati sono corretti, ma il risultato generale del sistema è alterato;
- Physical spoofing
 - Simulazione fisica della biometria.

Per garantire la sicurezza dei dati biometrici occorre adottare crittosistemi e relativi segreti a tutela del dato biometrico (Strong cryptiography and crypto-card based solutions).

La memorizzazione del dato biometrico è molto sensibile → occorre adottare tecniche per la sua difesa, in quanto una volta ruvato il dato non è rigenerabile.

17.6 FingerPrint

Le caratteristiche globali non sono sufficienti per il riconoscimento. Si utilizzano solo per classificare (clustering) le impronte (divisione in classi).

Le caratteristiche locali dette minuzie (singolarità) si utilizzano solo per il riconoscimento all'interno della classe prodotta dalle caratteristiche globali

17.6.1 Vantaggi e Svantaggi

Vantaggi	Svantaggi
Universalità	Efficienza (occupazione della memoria e tempo di ricerca)
Unicità	Stabilità (varia nel tempo)
Permanenza (Stabilità)	Insidia (relativamente semplice da duplicare)
Misurabilità	
Accettabilità	
Efficacia	
Acquisizione (Attiva e passiva) (basso costo dei sensori)	

17.6.2 Metodo di Matsumoto

Come detto l'impronta digitale è molto semplice da duplicare → basta considerare che con questo metodo basta un materiale plastico riscaldato su cui far aderire il dito per ottenere una duplicazione dell'impronta digitale (sono bassi anche i costi per far ciò)

17.7 Retina

È basato sull'acquisizione e la verifica della mappa vascolare della retina dell'occhio umano: presenta circa 180 caratteristiche misurabili.

La rete vascolare della retina è una delle caratteristiche che presenta un indice di variabilità nel tempo moderatamente basso.

È situata però in una posizione più interna rispetto all'iride, per cui i metodi per la scansione sono più invasivi.

17.7.1 Vantaggi e Svantaggi

Vantaggi	Svantaggi
Universalità	Efficienza (occupazione della memoria e maggiore cooperazione rispetto all'iride)
Unicità	Stabilità (alcune patologie potrebbero alterarla)
(Insidia) Spoofing possibile solo attraverso la chirurgia	Acquisizione (attiva e con un alto costo dei sensori)
Misurabilità	
Accettabilità	
Efficienza	
Efficacia	

17.8 Geometria della mano

Consiste nel riconoscimento delle persone mediante la verifica delle misure e della conformazione della mano e consente un discreto coefficiente di univocità

Caratteristiche:

- Forma;
- Locazione;
- Dimensione di mano, dita, nocche.

Misura le caratteristiche fisiche della mano:

- Lunghezza delle dita;
- Larghezza della mano;
- Spessore delle dita.

Le misure fisiche della mano sono catturate da un dispositivo CCD e la sagoma della mano viene memorizzata tridimensionalmente. La mano viene posta su un piano dove vi sono 5 pioli per posizionarla nel modo giusto. Vengono acquisite 2 immagini (dall'alto e di lato).

17.9 Firma

Tecnica riconosciuta ai fini dell'autenticazione.

Essa è soggetta ad un riflesso condizionato non facilmente riproducibile, soprattutto in real time.

La firma è utilizzata anche in processi di riconoscimento.

Si definiscono schemi di caratterizzazione per la dinamica dei movimenti durante la firma:

- Angolazione della penna;
- Pressione esercitata con la penna;
- Tempo di esecuzione;
- Velocità e accellerazione;
- Proprietà geometriche.

La firma presenta diverse criticità:

- Eccessiva variabilità intraclasse;
- Può essere dimenticata;
- Nella modalità di acquisizione offline contano solo le caratteristiche morfologiche (facili da duplicare);
- Nella modalità online (più complessa in quanto richiede HW speciale) si considerano altre caratteristiche più difficili da replicare, ma che possono essere variabili in base allo stato d'animo del soggetto che firma.

17.9.1 Vantaggi e Svantaggi

Vantaggi	Svantaggi
Universalità	Unicità
Misurabilità	Efficienza (occupazione della memoria e tempo di ricerca)
Insidia (difficile da duplicare nel caso online)	Insidia (facile da duplicare nel caso offline)
Accettabilità	Efficacia
Efficienza	Stabilità
Acquisizione (offline)	Acquisizione (online)

17.10 Riconoscimento vocale

Molto dipendente da una serie di fattori quali:

- Testo;
- Digitalizzazione del testo;
- Indipendenza dal testo.

17.10.1 Vantaggi e Svantaggi

Vantaggi	Svantaggi
Universalità	Unicità
Misurabilità	Efficienza (occupazione della memoria e tempo di ricerca)
Accettabilità	Insidia (facile da duplicare)
Acquisizione (attiva e passiva) (costo del dispositivo)	Acquisizione (modalità passiva)
	Efficiacia
	Stabilità

17.11 Riconoscimento dell'orecchio

17.11.1 Vantaggi e Svantaggi

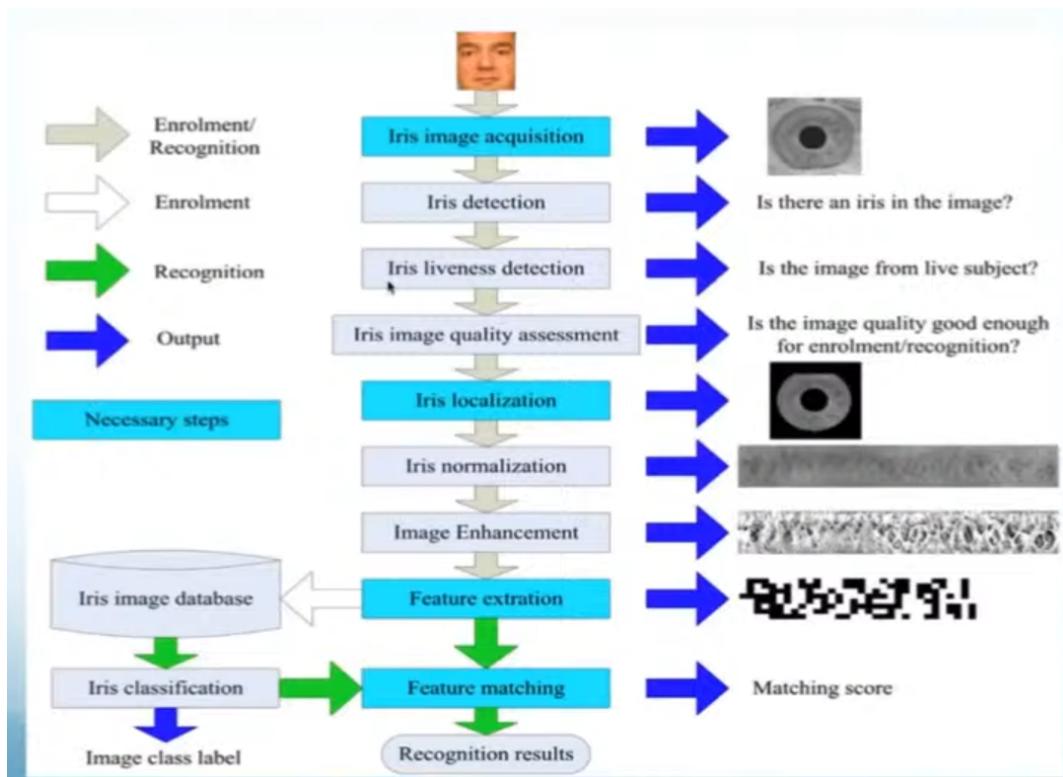
Vantaggi	Svantaggi
Universalità	Efficienza (occupazione della memoria e tempo di ricerca)
Unicità	Insidia (media difficolta nella duplicazione)
Stabilità	Acquisizione
Misurabilità	Accettabilità
Acquisizione	

17.12 Riconoscimento del volto

17.12.1 Vantaggi e Svantaggi

Vantaggi	Svantaggi
Universalità	Unicità
Efficienza	Efficacia (riconoscimento)
Efficacia (verifica)	Stabilità
Misurabilità	Insidia (facile da aggirare)
Accettabilità	
Acquisizione	

17.13 Iride



17.13.1 Polarizzazione dell'immagine

$$I(x(r, \Theta), y(r, \Theta)) \rightarrow I(r, \Theta)$$

Dove $x(r, \Theta)$ e $y(r, \Theta)$ sono definiti come combinazione lineare dei punti che delimitano la pupilla, cioè $x_p(\Theta)$ e $y_p(\Theta)$, e dei punti che delimitano la sclera cioè $x_s(\Theta)$ e $y_s(\Theta)$, come si può notare dalle seguenti equazioni:

$$x(r, \Theta) = (1-r)x_p(\Theta) + rx_s(\Theta)$$

$$y(r, \Theta) = (1-r)y_p(\Theta) + ry_s(\Theta)$$

Una volta che l'iride è stata localizzata, l'immagine viene trasposta in un sistema di assi polari (polarizzazione dell'immagine). Tale polarizzazione compensa la dilatazione della pupilla e la variabilità della dimensione, producendo una rappresentazione invariata rispetto a dimensione e traslazione. Da questa nuova immagine vengono quindi estratte le caratteristiche discriminanti dell'iride.

18 Lezione del 14-05

18.1 Iride - Matching

La fase di estrazione delle caratteristiche produce un *Iris Code*, ossia un codice di riferimento dell'iride di 256 byte, sulla base del quale viene effettuato il matching. Il confronto tra 2 Iris Code si effettua calcolando la distanza di Hamming tra i codici e dividendo il totale per il numero totale di bit ($N = 256 \times 8 = 2048$).

$$\text{HD} = \frac{1}{N} \sum_{j=1}^N A_j \otimes B_j$$

$$s(A, B) = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (A(i,j) - \bar{A})(B(i,j) - \bar{B})}{\sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (A(i,j) - \bar{A})^2 \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (B(i,j) - \bar{B})^2}}$$

18.2 Iride - nel dettaglio

La parte colorata dell'occhio umano, con le sue circa 300 caratteristiche misurabili, è forse l'unica caratteristica fisica maggiormente peculiare di un individuo:

- Non è passabile di cambiamenti nel corso del tempo;
- Non può essere modificata artificialmente;
- La probabilità di trovare sulla terra 2 iridi uguali è praticamente nulla (una su 10 seguito da 78 zeri);
- Ogni iride umano ha infatti una struttura unica al punto che persino l'iride destra e sinistra della stessa persona sono differenti.

18.3 Metodi di rilevazione dell'iride

Un sensore, collocato a circa 40 cm dalla persona esaminata, inizia a fotografare i margini dell'occhio; quindi attraverso un certo numero di scansioni successive, si individuano i contorni dell'iride come una corona circolare.

Poi si seleziona un quadratino alla volta di quest'area che rappresenta l'area da decodificare. Il disegno di questa regione viene poi convertito in un codice di 512 byte, il cosiddetto *Iris Code* che, confrontato con quelli archiviati nel database, identifica la persona.

18.4 NICE - Noisy Iris Challeng Evaluation

Competizione su un particolare dataset di immagini → lo scopo è quello di generare il miglior algoritmo di riconoscimento. Prevede 2 fasi:

- NICE.I → evaluated iris segmentation and noise detection techniques (97 partecipanti);
- NICE.II → evaluated encoding and matching strategies for biometric signatures.

Il protocollo prevedeva un applicativo che poteva essere scritto in qualsiasi linguaggio di programmazione, eseguito in una modalità standalone.

Inoltre era impedito qualsiasi accesso ad internet.

18.5 MICHE I & MICHE II

Prevede 4 fasi:

- Segmentation;
- Normalization;
- Coding;
- Matching.

18.6 Biometrie soft

È possibile studiare il movimento dello sguardo in presenza di un volto come biometria.

Le biometrie comportamentali (come camminata, firma, battitura tasti) sono legate al contesto in cui vengono rilevate e vengono abbinate a biometrie di tipo hard per aumentare l'accuratezza nei sistemi di riconoscimento biometrico.

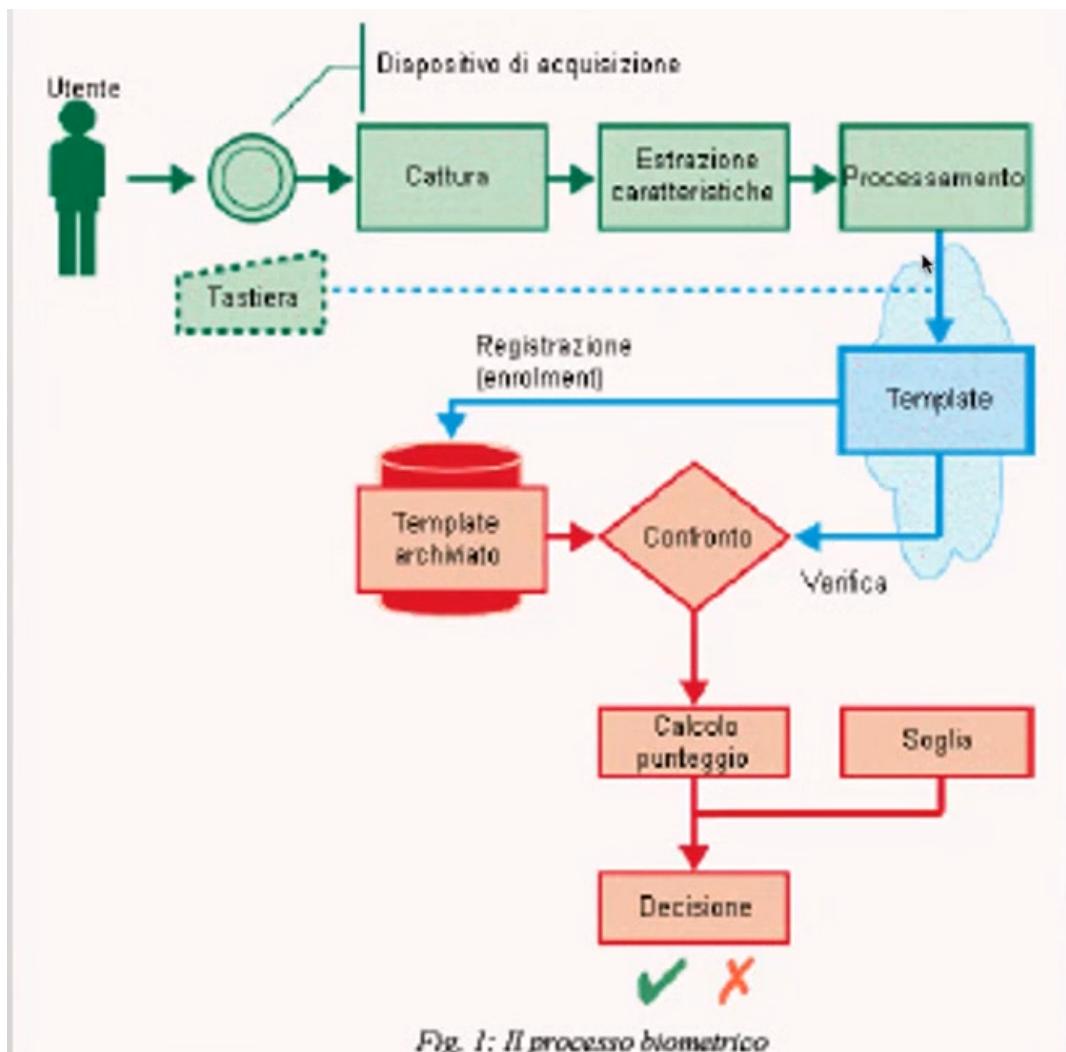
18.6.1 Tobii 1750

Corredato di telecamera e 5 diodi, sfrutta il riflesso corneale alla luce infrarossa. La pupilla rimane più luminosa e si può meglio delineare il contorno (fenomeno "bright pupil").

18.7 Ambiti di ricerca

- Analisi biometrica;
- Image processing;
- Financial Frecasting;
- Medical Image Anlaysis;
- Videosurveillance;
- IoT for Smart Home and Smart Cities;
- Object Tracking;
- Video Scenes Understanding.

18.8 Fasi di un tipico sistema biometrico



- Scelta della biometria;

- Verifica o Riconoscimento;
- Detection;
- Segmentation;
- Feature Extraction;
- Recognition.

18.9 Struttura di un riconoscitore biometrico

18.9.1 Fase di Enrollment

- Si associa un insieme di caratteristiche all'identità di un soggetto
 - Raccolta dei dati ed estrazione delle caratteristiche.
- Il modello estratto (template) è memorizzato in un database o su un supporto portatile (smart card);
- Questo processo può essere effettuato singolarmente o in blocchi (**batch enrollment**).

18.9.2 Fase di Testing

- Si estrae il template dal volto del soggetto che richiede di essere autenticato
 - Raccolta dei dati ed estrazione delle caratteristiche;
- Il modello estratto (template) è cercato in un database o su un supporto portatile (smart card);
- Un criterio di matching stabilisce se il soggetto deve essere autorizzato o no

18.10 Approcci alla detection

Tecniche Feature based → usano esplicitamente la conoscenza dell'aspetto del volto caratterizzato da un insieme di feature di basso livello:

- Low level Analysis → proprietà dei pixel
 - Bordi;
 - Colore della pelle;
- Feature Analysis → informazione sulla geometria del volto
 - Costellazione;
 - Feature Matching;
- Template Matching → modello standard definito manualmente o descritto da una funzione:
 - Correlazione;

- Snakes;
- ASM.

Tecniche Image based →

- Affrontano il problema della localizzazione come un generico problema di PR;
- L'obiettivo è imparare a riconoscere un'immagine sulla base di alcuni esempi di training.

18.11 Approcci alla recognition

Approcci Feature based → Prendono in input l'immagine originale e ne estraggono le caratteristiche distintive contenute al suo interno, (occhi, naso e bocca per il volto) e queste statistiche sono date in input ad un classificatore strutturale e Geometrico (Locali):

- PIE Issues.

Approcci Holistic → Utilizzano l'intera regione in esame come base per il riconoscimento (Globali).

Approcci ibridi.