

Appunti di

Architetture dei Calcolatori e Sistemi Operativi

Introduzione

Nelle pagine seguenti una sintesi delle dispense con arricchimenti del libro di testo di riferimento del corso.

Gli argomenti affrontati a lezione saranno:

- *Programmazione in C*
- *Programmazione concorrente*
- *Introduzione al RISC - V*
- *Chiamate a funzione in Assembly*
- *Componenti di base RISC-V*
- *Architettura RISC-V*
- *Architettura pipelined*
- *Ottimizzazione della Pipeline*
- *Struttura della memoria Cache*
- *Bus*
- *Kernel*
- *Scheduling*
- *Memoria virtuale*
- *Filesystem*
- *Driver*

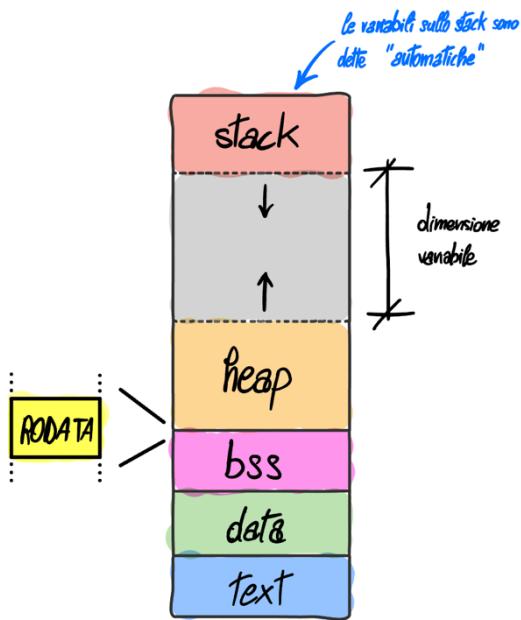
Bibliografia

Carlo Brandoles, *appunti e slides delle lezioni*, a.a. 2023/2024.

W. R. Stevens e S. Rago, *Advanced Programming in UNIX Environment*, 3^a Edizione.

Programmazione – Introduzione

La memoria RAM ha una struttura suddivisa in zone, come illustrato nello schema di seguito:



- **.text**, contiene codice eseguibile (sola lettura, dimensione fissa).
- **.data**, variabili statiche inizializzate.
- **.rodata**, valori costanti.
- **.bss**, dati statici, variabili globali/costanti, non inizializzati.
- **.stack**, non preallocata è una zona dinamica, gestita dal programma, contiene le variabili locali della funzione in esecuzione.¹
- **.heap**, memoria dinamica non ben definita (a differenza dello stack), più libera ma complicata da gestire.

Segmentation Fault: errore che si verifica quando lo stack cresce al di fuori della dimensione massima, solitamente indicato con SIGSEGV.

Puntatori a void: utilizzati per puntare a dati di qualunque tipo, sono necessari i cast dei dati.

Le operazioni di compilazione ed esecuzione di un programma, si suddividono in diverse fasi, generalmente complesse, alcune delle quali saranno descritte nei prossimi paragrafi. Si noti che tutte le operazioni e gli errori analizzati sono riferiti a sistemi UNIX e utilizzando il linguaggio C nella specifica C99.

Preprocessing: il risultato di quest'operazione è visibile utilizzando il comando seguente.

"gcc hello.c -E > hello.d"

file da compilare *file con direttive di preprocessing*

Assembly: linguaggio simbolico, risultato dell'operazione di compilazione, visibile attraverso il comando:

"gcc hello.c -S"

¹ Contiene anche parametri e valori di ritorno delle funzioni; inoltre, la dimensione dello stack cresce solo quando la funzione ne richiama altre oppure se ricorsiva.

Binario: è l'output della compilazione in codice binario, può essere letto aprendo il file oggetto con il programma ReadELF. Contiene tutti i segmenti che da spostare in RAM, dove viene indicato l'indirizzo iniziale per le zone Stack ed Heap.²

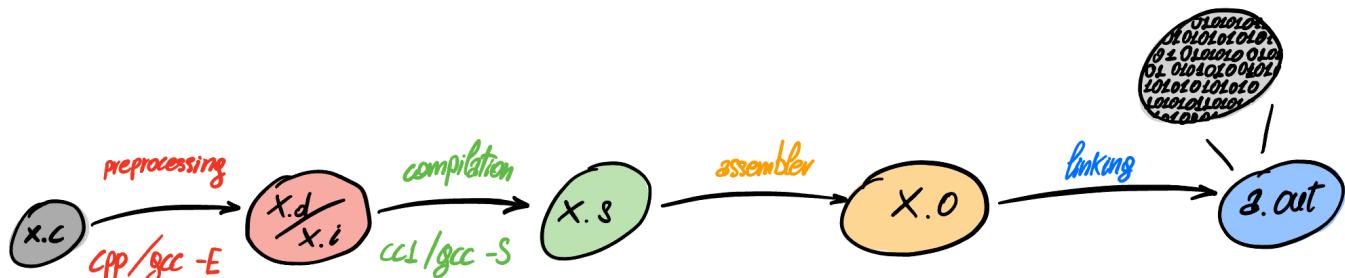
"*gcc hello.c -c*"

Eseguibile: il file oggetto vero e proprio, risultato della compilazione ed eseguibile dalla macchina, ottenuto col comando *gcc hello.c* al quale posso aggiungere poi un certo nome con *... -o name_exe*; di default sarà inserito nella *working directory* (cartella attuale), ma potrà essere specificato aggiungendo davanti al nome del file da compilare *./*.

Attributi delle variabili: sono opzionali e definiscono il comportamento della variabile, in totale ne analizzeremo quattro.

- Extern, specifica che la variabile è definita all'esterno.
- Volatile, specifica al compilatore che può cambiare durante l'esecuzione.
- Register, specifica al compilatore di inserire la variabile nei registri del processore.
- Static, specifica che una variabile mantiene il valore tra una chiamata e l'altra della funzione in cui è definita; è allocata in data, utilizzato soprattutto nelle librerie.

Il modello sottostante illustra le fasi che si susseguono nella compilazione di un programma in C, in particolare descrivendone l'output.



Makefile: insieme di regole che permettono di mantenere allineati più file collegati da dipendenze, aggiornando quelli obsoleti; qualunque regola è seguita dal comando *make*, mentre in GNU si usa *gmake*. La sintassi del *makefile* è la seguente.

<i>make</i>	$\left[\begin{array}{l} -C \text{ dir} \\ -f \text{ file} \\ -i \\ -j [n] \\ -n \\ -q \\ -t \end{array} \right]$	$[\text{goal}]$	<i>rifoma</i>	$\xrightarrow{\delta \text{ (successo)}}$	$1 \text{ (almeno un goal da aggiornare)}$
				$\xrightarrow{2 \text{ (errore)}}$	

² È presente un'apposita sezione per indicare al sistema operativo dove un segmento inizia e dove l'altro finisce; ai fini della comprensione, è assimilabile ad un header/descrittore.

- Esistono 5 tipologie di *goal*, in particolare: esplicite, implicite, definizione di variabili, direttive, commenti. La sintassi in questo caso sarà



Lo stile di programmazione da adottare è di tipo modulare, dove le funzionalità tra loro omogenee sono raggruppate in moduli, garantendo *information hiding*, *indipendenza tra moduli* ed *elevata coesione*³. Un modulo è formato da un file *header.h* per l'interfaccia ed uno o più file *source.c* ; si utilizza la compilazione condizionale per evitare di definire più volte una libreria, usando `#ifndef name.h #define`. Inoltre, si possono definire

- *Scope*, è la parte di codice dove il simbolo (variabile, costante o funzione) è visibile al compiler.
- *Linkage*, cioè l'area dove il simbolo è visibile al linker.

³ Si intende il grado di omogeneità tra le funzioni di un modulo.

Programmazione – Concorrenza

Concetto nato dall'esigenza di condividere le risorse di calcolo, dove si ha l'esecuzione di più programmi contemporaneamente; nel nostro corso si fa riferimento al parallelismo virtuale, non reale, con sistemi dotati di un solo processore (un core). Ciò introduce molti vantaggi ma anche una maggiore complessità nel sistema, dove il parallelismo sarà realizzato dall'alternanza dell'esecuzione di programmi mediante il sistema operativo.

include <unistd.h> obrena

Virtualizzazione: tecnica utilizzata dal sistema operativo per suddividere un a singola risorsa hardware, concettualmente assimilabile alle macchine virtuali, dove una “porzione” della risorsa è assegnata ad un programma in esecuzione evitando conflitti di accesso.

Processo: è l'istanza del programma in esecuzione, dinamicamente costruito dal sistema operativo; contiene il programma, le risorse virtuali ed informazioni utilizzate nella gestione del processo stesso dal sistema operativo.

- *PID*, identifica univocamente il processo, solitamente a 16bit
- *Process Descriptor*, struttura dati complessa che contiene informazioni di gestione

Nota: durante la compilazione viene assegnata la memoria virtuale, ma quando il programma viene eseguito, nel corrispettivo processo viene creata una mappa di memoria che associa *indirizzo virtuale* \Leftrightarrow *indirizzo fisico*. In questo modo due programmi con indirizzi virtuali identici non vanno in conflitto, garantendo la separazione della memoria.⁴

SWAP: area di memoria su disco dove viene salvato il contesto di un processo in attesa quando la RAM è piena; gestita dal sistema operativo.

Sui processi si possono compiere operazioni di *creazione* (processo crea processo, il primo è *init*, con PID = 1), *terminazione*, *attesa*, *segnalazione*, *sostituzione del codice*.

Multiprogrammazione: si ha un programma che crea più processi per sfruttare il parallelismo, dove il programma viene suddiviso in più parti che devono essere eseguite senza interruzioni.

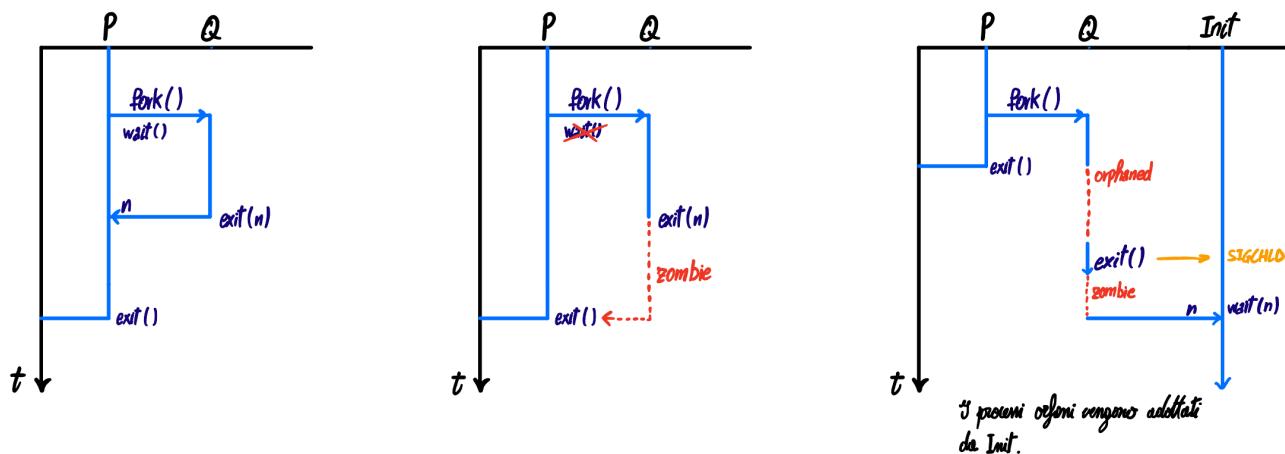
Nella pagina seguente un elenco delle principali funzioni utilizzate nella gestione dei processi.

⁴ Se il sistema dovesse fallire si avrebbe il *segmentation fault*.

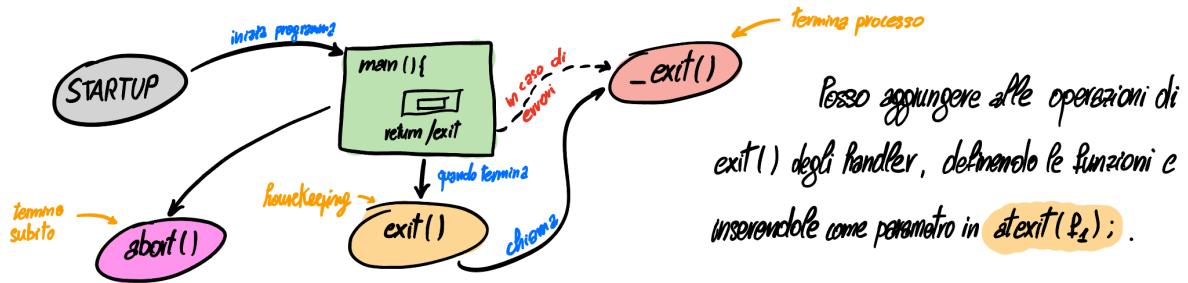
Gestione processi

- **exit (...)** funzione che termina processo,
→ parametri : valore di ritorno processo .
- **fork ()** funzione crea processo figlio, non ha parametri .
→ ritorno : 0 in figlio e pid figlio in padre .
- **wait (...)** funzione bloccante, attende
terminazione di un V processo figlio.
→ parametri : "status" → valore ritorno figlio
→ ritorno : pid figlio terminato, -1 errore
- **WIFEXITED (status), WEXITSTATUS (status)**,
sono due macro utilizzate per testare exit
status registrato dalle wait . La prima
e' vera se figlio termina naturalmente .
*s'è una funzione che
non ritorna, visto
che il codice viene
cominciato*
- **exec (), exec_p (...)** e' una famiglia di funzioni
useremo la exec_p(), sostituisce codice
processo in esecuzione con quello indicato.
→ parametro : nome programma da
sostituire.
- **signal_handler (...)** funzioni definite dall'utente per gestire
un segnale (e.g. SIGUSR1).
→ parametro : numero del segnale .
→ fun. auxil : signal(SIGUSR1, handler)
typedef void (*sigHandler_t)(int)
void Handler (int)
- **kill(...), raise(...), abort(...)** funzioni che lanciano segnali.
ne esistono altre

I processi seguono un certo ciclo di vita, che in caso di errori nella gestione può variare, di seguito i tre casi principali, il primo da sinistra corretto, mentre negli altri due sono illustrate delle condizioni di errore (Zombie e Orphaned).



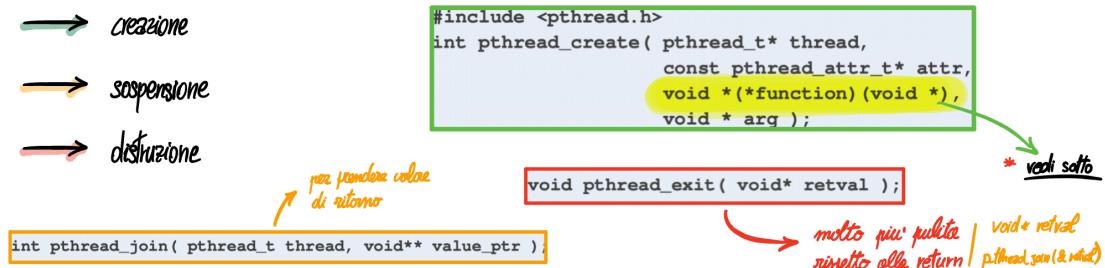
Segnali: utilizzati dai processi per compiere certe operazioni una volta ricevuti, solitamente terminano il processo, ma possono essere utilizzati per realizzare un “impreciso” meccanismo di sincronizzazione.



Per rispondere alle esigenze di parallelizzazione e sincronizzazione dei sistemi, sono stati introdotti i *thread*, anche detti processi leggeri.

Thread: unità d'esecuzione sequenziale (internamente) che può essere parallelizzata, si distingue dai processi perché esiste nel contesto di un processo e ne condivide la memoria con altri thread (creati dal processo). La sincronizzazione è ora a carico del programmatore che dovrà implementare appositi meccanismi.⁵

- Bypassa le operazioni bloccanti
- Identificato dal TID
- Su di esso si possono eseguire operazioni di



Funzione thread*: funzione che sarà eseguita dal thread avrà necessariamente il prototipo come il seguente.

`void* function (void*);`

Flusso di controllo: esecuzione sequenziale di istruzioni, realizzato da un thread mediante la funzione passata.

Nel corso si utilizzano i thread POSIX, dove “per costruzione” se un processo termina, allora terminano anche i suoi thread. Inoltre, le principali differenze tra *processi* e *thread* possono essere riassunte e descritte dalla tabella nella pagina successiva.

⁵ Qualunque thread ha il proprio stack, anche se più piccolo; inoltre, un thread può creare un altro thread.

	Processi	Thread
Creazione Discrizione	Richiedono allocazione, copia e deallocazione di grandi quantità di memoria	Richiedono solamente la creazione di uno stack per il thread
Errore	Non può danneggiare altri processi	Può danneggiare altri thread e l'intero processo cui appartiene
Codice	Un processo può modificare il proprio codice mediante il cambiamento di eseguibile	Il codice di un thread è fissato e presente nella sezione text del processo cui appartiene
Condivisione	E' onerosa e deve essere implementata dal programmatore	E' automaticamente garantita poiché tutti i thread condividono la memoria del processo cui appartengono
Mutua esclusione	La mutua esclusione è garantita automaticamente dall'isolamento proprio dei processi	Deve essere realizzata dal programmatore mediante semafori, mutex, ecc
Prescrizioni	Limitate dall'overhead di gestione	Elevate
Concorrenza	Limitata dalla difficoltà di comunicazione	Elevate

Mentre le principali funzioni utilizzate nella gestione dei thread sono le seguenti.

Gestione Thread

- `pthread_join(...)` funzione usata per attendere terminazione thread.
 - parametro 1: TID da attendere.
 - parametro 2: *void → prende valore di uscita del thread.
- `pthread_mutex_lock(...)` funzione per richiedere lock su mutex, è bloccante (variante `trylock no`).
 - parametro: puntatore mutex.
- `#include <semaphore.h>` libreria usata per i semafori (sincronizzazione).
 - `sem_t semaloro;` oggetto semaforo, usato come contatore.
 - `sem_init(...)` funzione per inizializzare semaforo.
 - parametro 1: &semaforo, punta a sem.
 - parametro 2: sempre '0'.
 - parametro 3: valore con cui inizializzare il semaforo.
 - `sem_destroy(...)` funzione per deinitializzare semaforo.
 - parametro: puntatore a semaforo.
 - `sem_wait / trywait / post (...)` funzioni analoghe a quanto visto per le mutex `lock / trylock / unlock`.
 - parametro: puntatore a semaforo.

In particolare, il MUTEX può essere visto come un gettone che viene richiesto da una funzione per essere eseguita, è un meccanismo di sincronizzazione usato nei thread, quando si vuole che due o più thread diversi non modifichino simultaneamente un dato globale (generalmente).



Mentre, il Semaforo è un ulteriore meccanismo di sincronizzazione che è invece un “contenitore di gettoni” dove se il valore è 0 il sistema si sospende in attesa che venga incrementato; quando una funzione esegue toglie il token, e lo rimette appena termina.

Nel nostro caso si utilizza il semaforo binario, particolarmente utile per gestire situazioni produttore/consumatore, dove si utilizzano due oggetti semaforo, uno attivo quando viene prodotto un dato ed un altro attivo quando può essere consumato il dato. Risulta molto simile al MUTEX ma molto più flessibile.

RISC-V – Introduzione

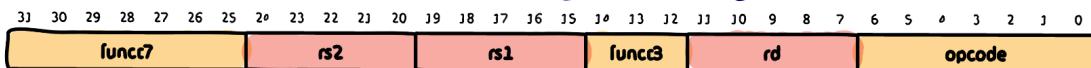
Instruction Set Architecture: abbr. ISA, insieme di istruzioni che definiscono una certa architettura.

Le architetture si distinguono in base all'ISA, in particolare si distinguono le CISC e le RISC⁶, la prima più complessa ma che permette di implementare più operazioni, mentre la seconda è più semplice ed economica.

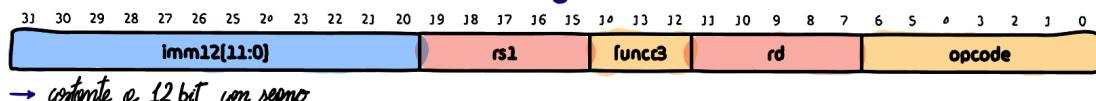
Sarà analizzato il processore di tipo RISC – V (five) con architettura a 32 bit, dove le istruzioni si distinguono in base alla codifica ed ai tipi di indirizzamento; inoltre è caratterizzato dall'avere 32 registri numerati da 0 a 31 su 5 bit. Prevede 4 formati con due varianti di istruzioni dove l'operazione da svolgere è sempre definita da un **OPCODE** ma anche da **funct3** e/o **funct7** (sono opzionali). Questi identificatori si trovano sempre alla stessa posizione (ultimi due se presenti).

Modelli e struttura delle istruzioni: di seguito una modellizzazione della struttura delle istruzioni utilizzate nell'architettura RISC-V.

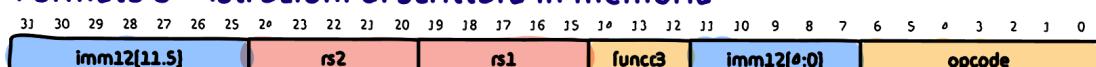
Formato R – Istruzioni aritmetico-logiche fra registri



Formato I – Istruzioni aritmetico-logiche immediate



Formato S – Istruzioni di scrittura in memoria



→ costante a 12 bit con segno.

Formato U – Istruzioni sulla parte alta dei registri



→ costante a 32 bit, con segno e 12 bit impliciti a destra.

Formato B – Variante di S – Istruzioni di salto condizionale



→ costante a 13 bit con segno e zero implicito a destra.

⁶ Rispettivamente sono il Complete Instruction Set Architecture ed il Reduced Instruction Set Architecture.

Formato J – Variante di U – Istruzioni di salto non condizionale



→ costanti a 21 bit, un segno ed uno zero implicito a destra.

#	Nome	Uso
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary registers Caller-saved
x6	t1	
x7	t2	
x8	s0 (fp)	Saved registers (Frame pointer) Callee-saved
x9	s1	
x10	a0	Function arguments Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers Callee-saved
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporary registers Caller-saved
x29	t4	
x30	t5	
x31	t6	

Registri: i registri dell’architettura sono riassunti e descritti nella tabella a fianco, dove si può osservare che il program counter può essere considerato come tale, ma non si può “utilizzare” (dal punto di vista del programmatore).

Osservazione: è buona norma suddividere il codice in due sezioni,

- **.data**, che contiene la parte di “dichiarazione delle variabili”, cioè l’inizializzazione dei registri.
- **.text**, contiene il codice vero e proprio.

Nelle pagine successive un elenco delle istruzioni di base del linguaggio Assembly del RISC-V, che sarà utilizzato nelle esercitazioni.

Oltre che la realizzazione di semplici programmi, gli esercizi verteranno generalmente sulla conversione di programmi o porzioni di codice scritti in C in Assembly: operazione che risulterà più semplice andando ad utilizzare dei costrutti di base per implementare i cicli o le condizioni, adattandoli caso per caso.

Istruzioni aritmetico/logiche registro-registro

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	Add	R	0110011	0x0	0x00	$rd = rs1 + rs2$
sub	Sub	R	0110011	0x0	0x20	$rd = rs1 - rs2$
xor	Exclusive or	R	0110011	0x4	0x00	$rd = rs1 \ ^ rs2$
or	Or	R	0110011	0x6	0x00	$rd = rs1 rs2$
and	And	R	0110011	0x7	0x00	$rd = rs1 \& rs2$
sll	Shift left logical	R	0110011	0x1	0x00	$rd = rs1 << rs2$
srl	Shift right logical	R	0110011	0x5	0x00	$rd = rs1 >> rs2$
sra	Shift right arithmetic	R	0110011	0x5	0x20	$rd = rs1 >> rs2$
slt	Set if less than	R	0110011	0x2	0x00	$rd = (rs1 < rs2)?1:0$
sltu	Set if less than (U)	R	0110011	0x3	0x00	$rd = (rs1 < rs2)?1:0$

Istruzioni aritmetico/logiche registro-immediato

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
addi	Add Immediate	I	0010011	0x0		$rd = rs1 + imm$
xori	Exclusive or immediate	I	0010011	0x4		$rd = rs1 \ ^ imm$
ori	Or immediate	I	0010011	0x6		$rd = rs1 imm$
andi	And immediate	I	0010011	0x7		$rd = rs1 \& imm$
slli	Shift left logical immediate	I	0010011	0x1	imm[5:11]=0x00	$rd = rs1 << imm[0:4]$
srl	Shift right logical immediate	I	0010011	0x5	imm[5:11]=0x00	$rd = rs1 >> imm[0:4]$
srai	Shift Right arithmetic immediate	I	0010011	0x5	imm[5:11]=0x20	$rd = rs1 >> imm[0:4]$
slti	Set less than immediate	I	0010011	0x2		$rd = (rs1 < imm)?1:0$
sltiu	Set less than immediate	I	0010011	0x3		$rd = (rs1 < imm)?1:0$

Istruzioni di trasferimento memoria-registro (load)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$
lh	Load Half-Word	I	0000011	0x1		$rd = M[rs1+imm][0:15]$
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$

Istruzioni di trasferimento registro-memoria (store)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$
sh	Store Half-Word	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$

Istruzioni di salto condizionato

sono pc-relative

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
beq	Branch if equal	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	Branch if not equal	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	Branch if less than	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	Branch if greater or equal	B	1100011	0x5		if(rs1 >= rs2) PC += imm
bltu	Branch if less than (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm
bgeu	Branch if greater or equal (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm

Istruzioni di salto incondizionato

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
jal	Jump and link	J	1101111			rd = PC+4; PC += imm
jalr	Jump and link register	I	1100111	0x0		rd = PC+4; PC = rs1 + imm

Altre istruzioni

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
lui	Load upper immediate	U	0110111			rd = imm << 12
auipc	Add upper immediate to PC	U	0010111			rd = PC + (imm << 12)

Altre istruzioni

estensione M

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	Multiply	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	Multiply high	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	Multiply high (S x U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	Multiply high (U x U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	Divide	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	Divide (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

Calcolo offset salto condizionato: l'indirizzo di destinazione di un salto condizionato è specificato come $offset = \frac{address(label) - pc}{2}$, espresso in [half-word] con costanti a 12 bit.⁷ Il range di salto è il seguente.

- *Istruzioni*, $[-2048; +2047] \sim \pm 2Kinsn$
- *Bytes*, $[-4096; +4094] \sim \pm 4KBytes$

Calcolo offset salto incondizionato: come il precedente, si differenzia per essere espresso da costanti a 20bit (con bit隐式) e per avere i seguenti range di salti.

- *Istruzioni*, $[-524288; +524287] \sim \pm 512Kinsn$
- *Bytes*, $[-1048576; +1048575] \sim \pm 1MBytes$

Calcolo offset variabili globali: l'indirizzo delle variabili globali è specificato come $offset = address(label) - pc$, si esprime in bytes con costanti a 12bit dove il valore dell'offset è quello della costante; il range di indirizzamento è il seguente.

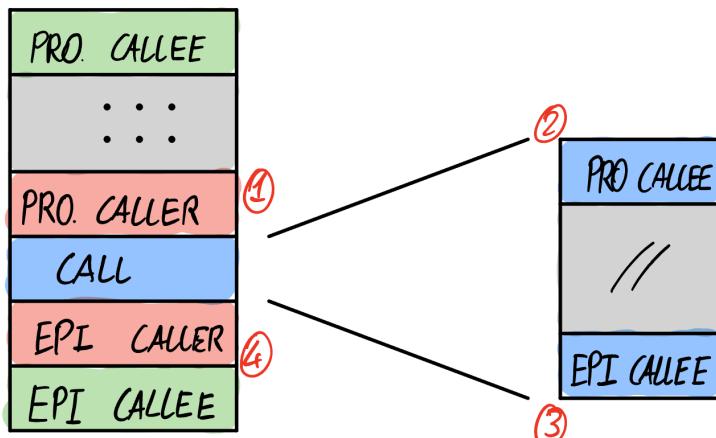
- *Bytes*, $[-2048; +2047] \sim \pm 2KBytes$

Istruzione di load address: l'istruzione *la* viene espansa in *auipc* e *addi*, come nell'esempio di seguito, dove $la\ x1,B \Rightarrow auipc\ x1, \Delta B[31;12] \quad addi\ x1,x1,\Delta B[11;0]$.

⁷ Il bit隐式 a destra rende conto della divisione per due.

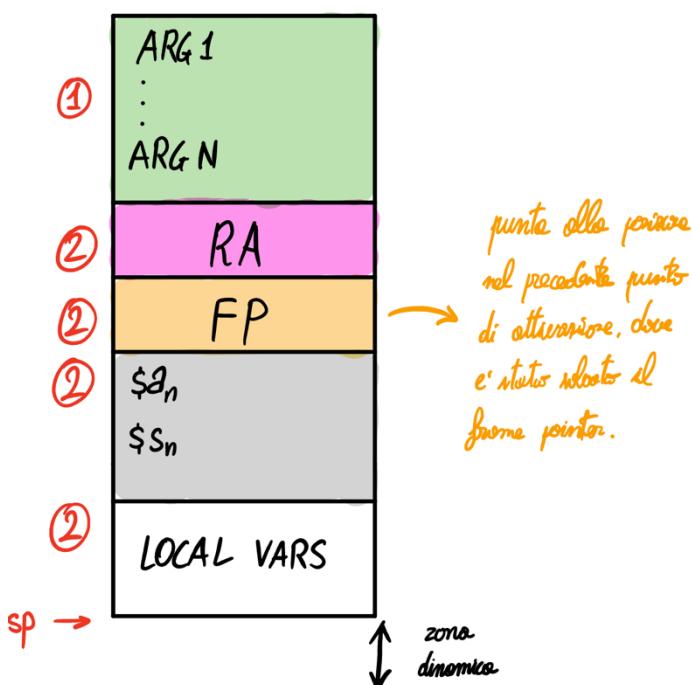
RISC-V – Chiamate di funzione ASM

La struttura generale utilizzata nella chiamata di una funzione è la seguente:⁸



- Prologo chiamato (trad. callee)
- Prologo chiamante (trad. caller)
- Chiamata
- Epilogo del chiamante, al ritorno della funzione
- Epilogo del chiamato

Stack: la struttura generale prevede la composizione del modello sottostante, dove gli argomenti della funzione sono solitamente salvati in ordine inverso. Sono indicati anche i punti dove viene modificato lo stack dalla chiamata di funzione. In particolare, viene rappresentata l'operazione di allocazione dello stack.

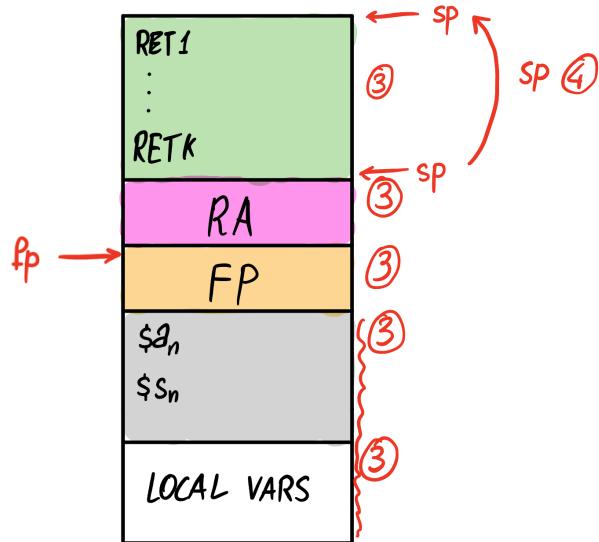


I “parametri” (nell’ipotesi che non debbano variare) vanno passati con i registri appositi, non si inseriscono nella sezione arg1-argN, salvandoli nello stack insieme ai registri di tipo saved. Una volta terminata l’allocazione, lo stack pointer punta al primo indirizzo utilizzabile dopo la fine della sezione *local variables*, dove possono essere contenuti i registri temporanei.

Per individuare i parametri della funzione si usa come riferimento la posizione del frame pointer (fissa).

⁸ I punti numerati indicano dove (ed in che ordine) viene manipolato lo stack.

Mentre la de-allocazione dello stack avviene a partire dall'epilogo del chiamato (3), come mostrato di seguito. Smonto lo stack riportando temporaneamente lo stack pointer all'inizio del campo ARG1-N (sarà cancellato, al suo posto i valori di ritorno); dopodiché aggiungo i valori di ritorno, che saranno presi dalla funzione chiamante (*hp*. ci troviamo nella funzione chiamata).



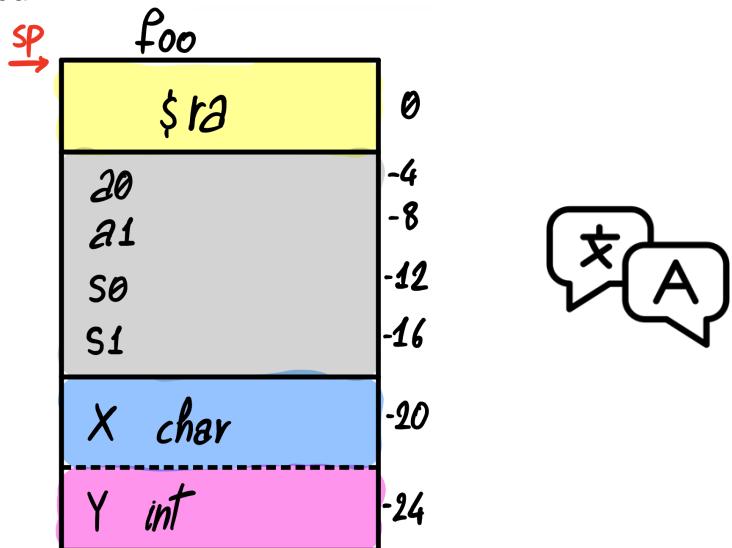
I precedenti erano casi generali con le convenzioni solitamente utilizzati nell'architettura di un processore di tipo RISC-V, ora verranno analizzati i casi specifici più semplici.

Push e Pop: sono le due operazioni che si possono eseguire sullo stack, dove nel primo caso si scrive e poi si incrementa, mentre il pop è il duale del precedente.

Esempio: di seguito la traduzione di un programma C in codice assembly.

<pre> int v; int inc(int x); main(){ v=inc(12); } </pre>		<pre> .data v: .space 4 .text main: addi a0, x0, 12 jal ra, inc la t0, v sw a0, 0(t0) inc: addi a0, a0, 1 jr ra .end </pre>
---	--	---

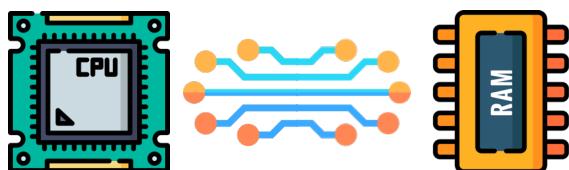
Un ulteriore esempio è la costruzione dello stack per una funzione con due parametri in input.



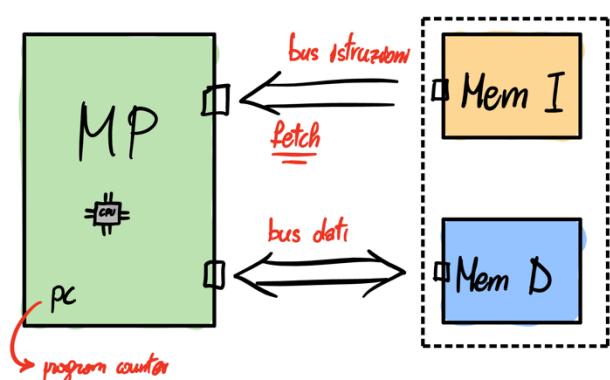
```
foo:    sw      ra, 0(sp)
        sw      a0, 4(sp)
        sw      a1, 8(sp)
        sw      s0, 12(sp)
        sw      s1, 16(sp)
        addi   sp, sp, -24
                #function body here
        addi   sp, sp, +24
        lw      ra, 0(sp)
        lw      a0, -4(sp)
        jr      ra
.end
```

RISC-V – Componenti

L'architettura, nel suo caso più semplice, è formata da un microprocessore e dalla memoria, interconnesse da un bus⁹.



È anche possibile vedere il microprocessore con un'altra struttura per comprenderne meglio il funzionamento; l'esecuzione di un'istruzione è un insieme di differenti operazioni, di seguito sono descritte quelle dell'architettura RISC-V.



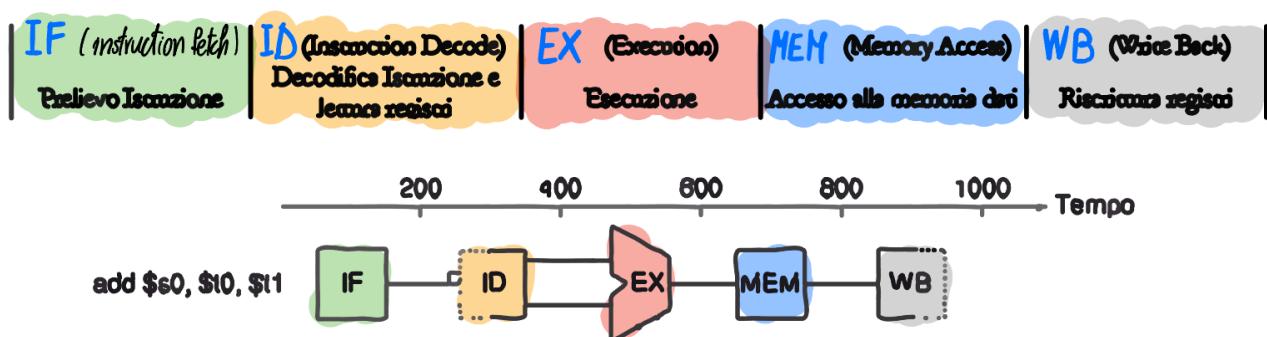
Fetch: prima fase, lettura dell'istruzione dalla memoria delle istruzioni e caricamento nel processore della stessa.

Decode: interpretazione dell'istruzione letta nella fase di *fetch*, accede ai dati in ingresso.

Execute: fase di esecuzione vera e propria.

Memory: accesso alla memoria dati.

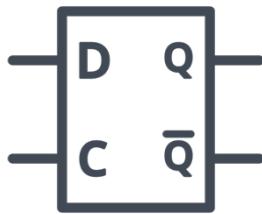
WriteBack: è la fase di scrittura del risultato; inoltre, viene incrementato il program counter, così che venga eseguita la prossima istruzione.



Ma come è fatto un microprocessore? Con quali componenti?

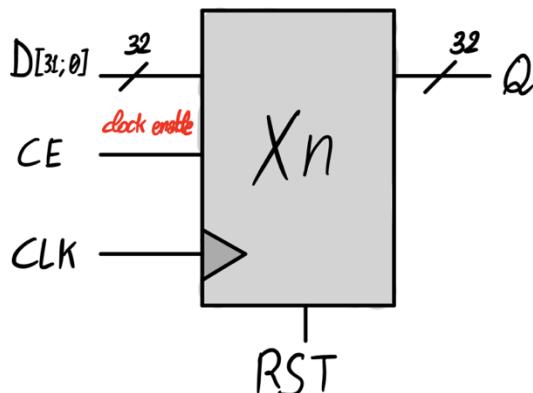
⁹ Il bus è un sistema di comunicazione costituito da più linee di comunicazione digitale, sarà approfondito più avanti nell'apposito capitolo.

Registri: un registro è una memoria formata da uno o più *flipflop* di tipo D, dove il contenuto del registro può variare solo quando arriva il segnale di *clock* (C).

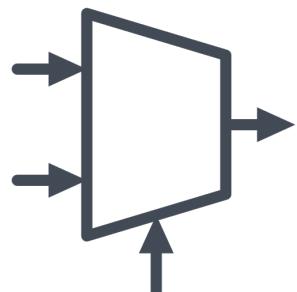


È anche presente un segnale di reset, solitamente indicato con RST, che viene attivato per azzerare il contenuto del registro.

Nel RISC-V si utilizzano registri a 32 bit composti da 32 Latch di tipo D. Di seguito una rappresentazione del modello (a 32bit).

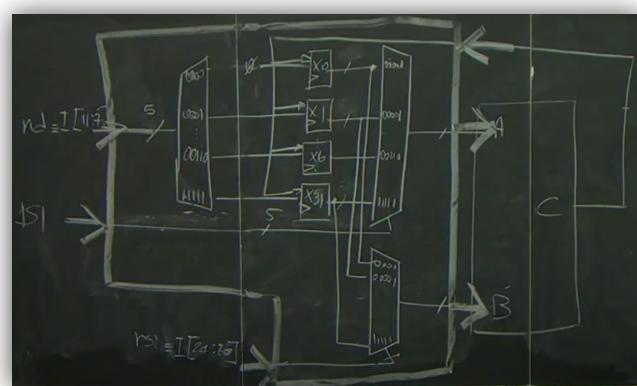


Multiplexer: utilizzando un circuito di base come esempio di funzionamento, sceglie quale segnale di ingresso mandare in uscita, in funzione di un apposito segnale di controllo. Nel nostro caso sono utilizzati nelle versioni a 32 bit.



Quando viene prelevata una certa istruzione, i bit dei campi delle istruzioni sono presi per instradare l'istruzione stessa nella "direzione" corretta.

Register file: contiene tutti i registri del processore, 32 nel caso dell'architettura RISC-V in esame; è proprio utilizzando il multiplexer (o un insieme di MUX, si veda il precedente paragrafo) che si rende possibile la scelta del registro da utilizzare.



10

¹⁰ Struttura interna del Register File, con registri e multiplexer/decoder.

Osservazione: non è possibile inviare ad un preciso registro un dato senza che lo prendano anche gli altri (in ingresso), però si può fare in modo che gli altri registri ignorino gli ingressi con il segnale CE (clock enable), che sarà attivo solamente sul registro su cui dovrà essere mandato il dato. È possibile scegliere “dove” attivare il CE utilizzando un Decoder.

Ogni formato di istruzione compie differenti operazioni, infatti, quelle R/I/U/J scrivono sempre (sui registri), mentre le S/B non scrivono.

Come si può comunicare al sistema che se i formati delle istruzioni sono S o B non deve scrivere alcun dato?

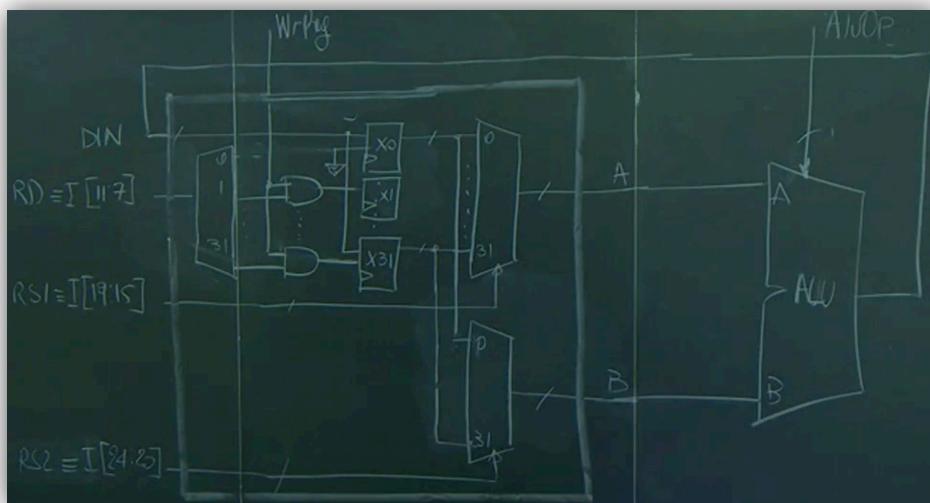
RISC-V – Architettura

WrReg: o *write register*, è un segnale di controllo utilizzato insieme alle porte AND, che ricevono in ingresso sia il *WrReg* che il segnale di CE diretto ai registri, se entrambi a 1 allora verrà lasciato passare il segnale di *clock enable* altrimenti no; in questo modo si specifica al sistema di non scrivere sui registri nel caso delle istruzioni S e B.

Control Unit: è il componente che riceve in ingresso l'istruzione da eseguire e, nel caso in cui non sia di tipo S o B, setta il *WrReg* a 1, altrimenti a 0 (S/B).¹¹

ALU: l'*Arithmetic Logic Unit* è il componente che si occupa di calcolare le normali operazioni aritmetiche, ciascuna scelta in base al valore dei bit nei campi *funct* (e.g. *funct3*, *funct7*). A livello fisico è un blocco che contiene diversi moduli, uno per ciascuna operazione, che calcola tutti i risultati possibili e sceglie poi di portare in uscita (utilizzando un MUX, controllato dal segnale *AluOp*) quello relativo all'operazione desiderata.

L'architettura così creata, fino a questo punto (non è ancora completa, risulta essere la seguente.

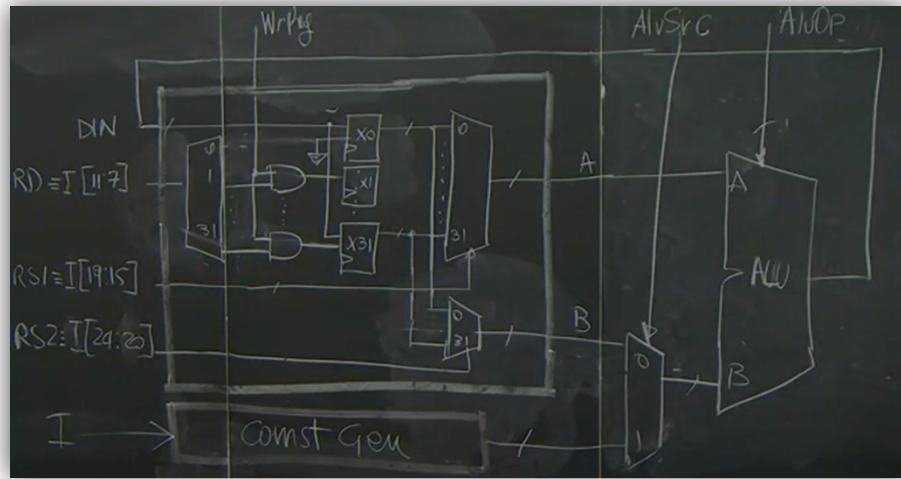


ConstGen: è un componente necessario per generare costanti, riceve in ingresso l'intera istruzione; questo perché sulla base del formato (dell'istruzione) decide che bit prendere per costruire la costante da utilizzare.

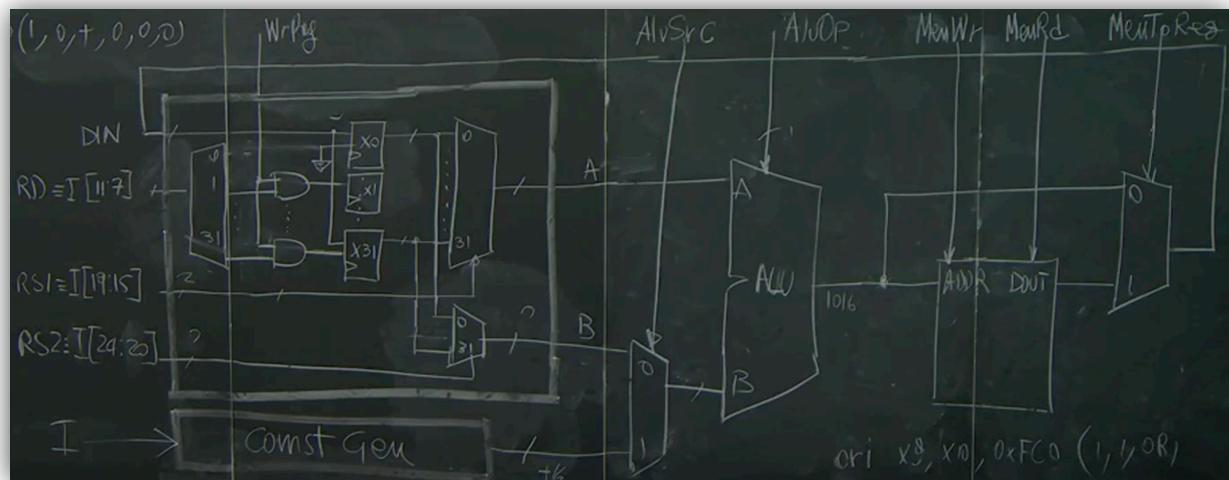
AluSrc: *ALU source* è un segnale di controllo, utilizzato per controllare il MUX che indica la sorgente della ALU, vale 0 per le istruzioni di formato R, 1 altrimenti ($\neq R$).

Anche il componente che calcola il segnale *AluSrc* può essere facilmente sintetizzato utilizzando le tecniche utilizzate nella sintesi delle reti combinatorie, l'architettura aggiornata sarà la seguente.

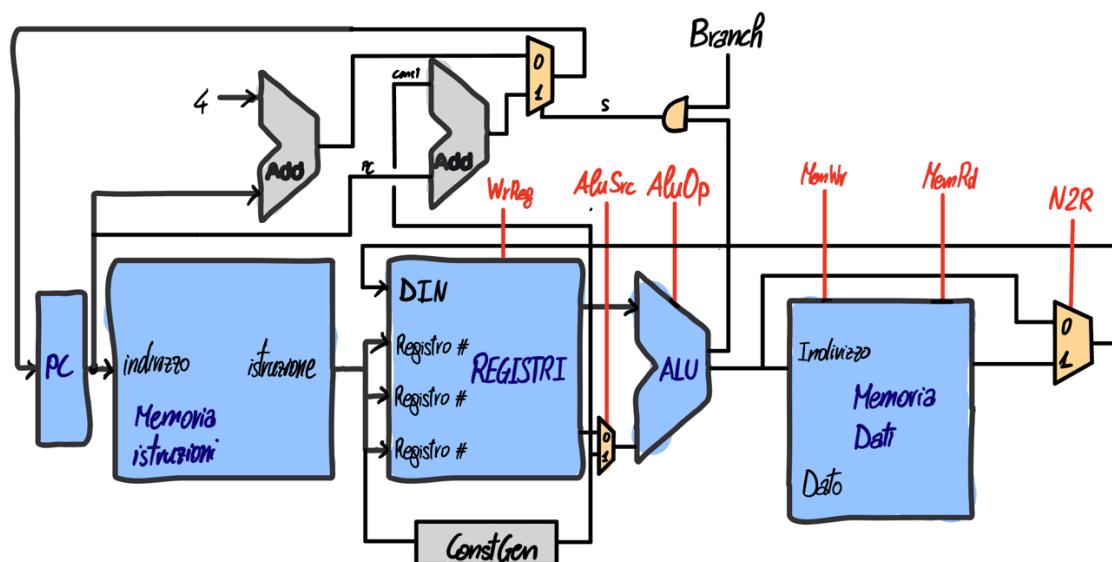
¹¹ È un componente che può essere sintetizzato come visto con le funzioni nel corso di Reti Logiche.



Aggiungendo anche i componenti ed i segnali necessari per le istruzioni *lw*, il precedente modello si arricchirà comprendendo:



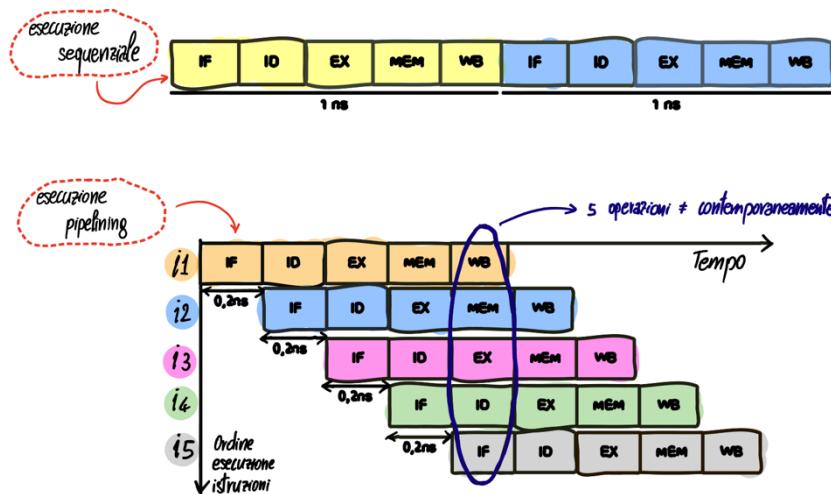
Nota: anche in questo caso si può osservare che i dati vengono portati “ovunque”, ma viene scelto quello richiesto utilizzando gli opportuni segnali di controllo; un funzionamento assimilabile ad una rete combinatorio costituita da MUX.



¹² È lo schema dell'architettura di riferimento per il RISC-V a singolo ciclo (no pipelined).

Architettura – Pipelining

Pipelining: è una tecnica di ottimizzazione per le strutture hardware, che idealmente permette di completare un'operazione complessa ad un rate di "1 per clock".



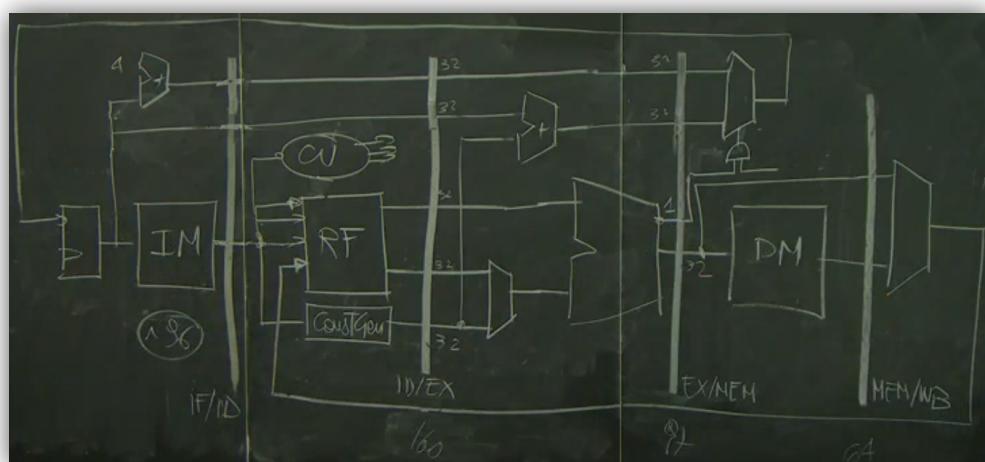
A sinistra il confronto tra ciò che avviene in un'architettura "standard" (sopra) ed una di tipo *pipelined*, dove per ogni ciclo di clock possono essere eseguite 5 operazioni nello stesso ciclo di clock.

Si noti che tutte le operazioni devono essere diverse, non posso eseguire allo stesso tempo due *fetch* o due *execute*.¹³

Nonostante i vantaggi presentati non è esente da problematiche, l'architettura pipelined introduce conflitti generalmente riguardanti l'accesso ai dati.

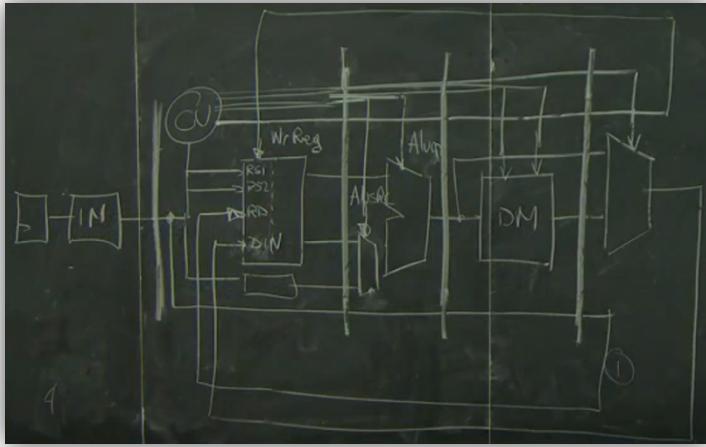
Conflitti: in breve, i conflitti generati sono di tre tipologie.

- *Strutturali*, introdotti dalla struttura della pipeline, facilmente risolvibili. (non saranno presi in considerazione negli esercizi).
- *Conflitti sui dati*, come l'utilizzo di un risultato prima che sia effettivamente pronto.
- *Conflitti di controllo*, tentare la scelta della prossima istruzione da eseguire prima che la condizione sia effettivamente valutata.



¹³ In generale migliora di 5 volte la velocità di esecuzione delle istruzioni, sicuramente più conveniente rispetto un'architettura a singolo ciclo.

Per limitare i conflitti sui dati si può modificare l'architettura, propagando i segnali per rendere disponibili i dati alle fasi precedenti, senza aspettare che siano scritti negli appositi registri (vengono "propagati all'indietro"); di seguito il nuovo schema dell'architettura che evidenzia le propagazioni dei dati, delle istruzioni e segnali di controllo.

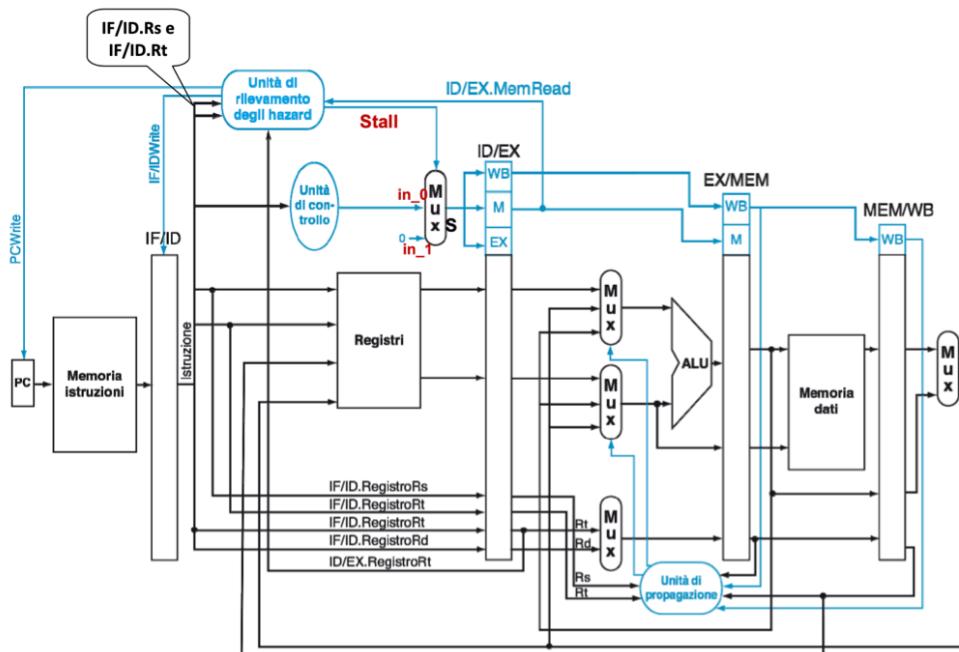


Osservazione: istruzioni, segnali di controllo e dati devono "viaggiare" insieme, per questo sono usate apposite unità di propagazione nell'architettura pipelined del RISC-V.

Convenzioni: quando ci si riferisce al "tempo" a meno che non sia esplicitamente indicato, si intende il tempo discreto (per non complicare eccessivamente schemi e grafi).

Come si sceglie la frequenza di clock?

La pipeline non è bilanciata, questo perché ogni stadio ha la sua durata. Si sceglierà allora come frequenza di clock quella della fase/componente più lenta.¹⁴ Lo schema sottostante è preso dalle slide che fanno riferimento al MIPS, è stata utilizzata per dare un'idea di massima dell'architettura completa.



¹⁴ Questo "ragionamento" è applicabile anche in altre situazioni dove è richiesto di scegliere una frequenza di clock adatta al sistema; la massima frequenza utilizzabile sarà quella minima di funzionamento del componente più lento. (e.g. Prova Finale di Reti Logiche)

Soluzioni per conflitti sui dati: si inseriscono istruzioni *NOP* (nulle), oppure si possono riordinare le istruzioni senza cambiare il “significato” del programma; in quest’ultimo caso, il compilatore inserisce tra due istruzioni dipendenti una istruzione indipendente/*NOP*. Fino ad ora, le soluzioni proposte non prevedono modifiche all’architettura hardware del RISC-V *pipelined*.

Un ulteriore tecnica (hardware) è invece la propagazione in avanti nei dati (*bypassing*) oppure o l’inserimento di stalli nella pipeline, che bloccano solamente una fase, non tutte le operazioni per 5 cicli.

Soluzioni per conflitti di controllo: nel caso in cui si abbia ritardo nel determinare la soluzione da prelevare (in seguito a condizione), se si utilizza una pipeline

- *Standard (senza predizione)*, è sufficiente inserire 3 NOP dopo ogni istruzione di salto condizionato (SW solution) oppure 3 stalli di *Fetch* dopo un salto condizionato (HW solution).
- *Standard (con predizione)*, predice sempre che il salto non venga eseguito (*untaken branch*), quando dovrà essere eseguito si dovranno scartare 3 istruzioni già presenti in pipeline.
- *Ottimizzata (senza predizione)*, richiede anticipazione del calcolo dell’indirizzo di destinazione e valutazione della condizione per il salto; i possibili problemi sono la necessità di propagazione ed i conflitti sui dati.
- *Ottimizzata (con predizione)*, si predice sempre *untaken branch* e si genera uno stallo quando il salto deve essere eseguito, scartando un’istruzione già in pipeline (segnali *ctrl = 0*).

Nota: si noti che nella pagina precedente è presente uno schema in che contiene più componenti e link di quelli descritti finora; questi saranno descritti nel prossimo capitolo che riproporrà, per semplicità, lo stesso modello.

Architettura – Ottimizzazione

Sarà analizzato il modo in cui il RISC-V risolve i conflitti ed esegue le operazioni nel modo più efficiente (ottimizzato) possibile. Il “momento” in cui il sistema identifica un conflitto è nello stadio di *decode*.

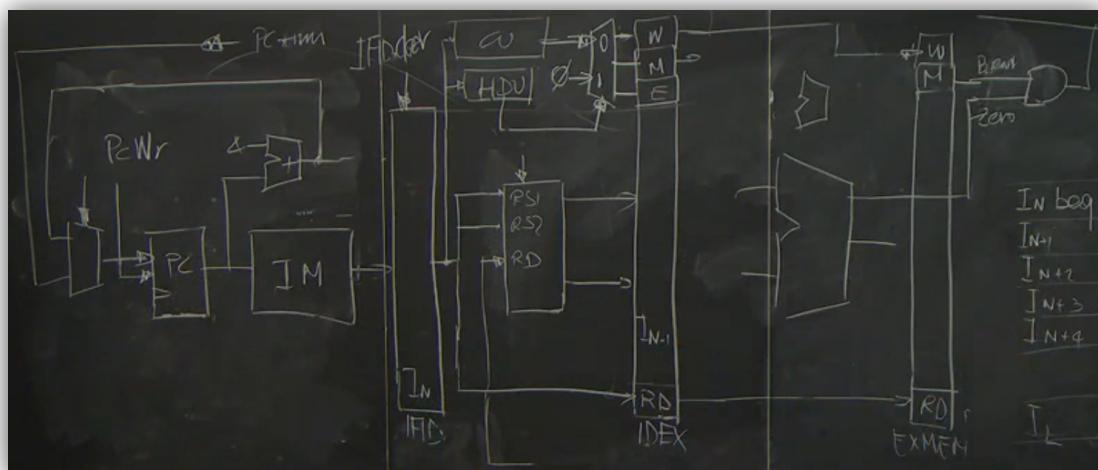
NOP instruction: istruzione che non da’ alcun risultato, andrà ad occupare tutti gli stadi di pipeline; è utilizzata per far sì che i dati precedenti vengano scritti nei registri prima di essere utilizzati dall’istruzione successiva.



Stallo: anche detto “bolla”, è l’introduzione di un ciclo di clock che non produce alcun risultato, di fatto va’ ad introdurre un ritardo per far sì che i dati da utilizzare nell’istruzione corrente vengano scritti nei registri dalla precedente.

Hazard Detection Unit: o HDU, è il componente che si occupa di identificare i conflitti nella *decode*; l’identificazione avviene semplicemente verificando se l’istruzione attuale ha il registro sorgente uguale al registro destinazione dell’istruzione precedente.

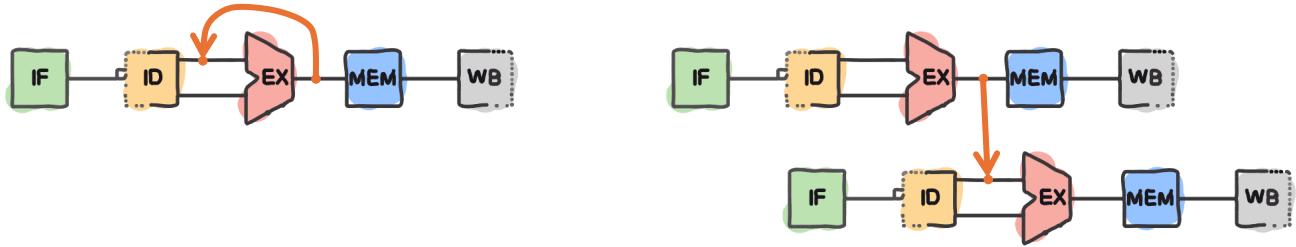
$$R_{source}(I_n) = R_{dest}(I_n - 1) \Rightarrow Hazard$$



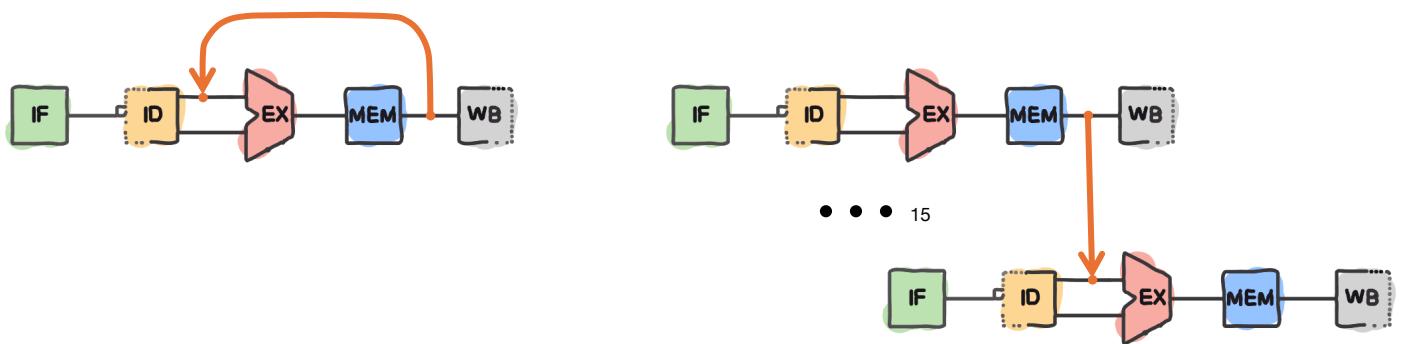
Politiche di branch: sarà utilizzata come politica di predizione dei salti la *branch untaken*, cioè si predice sempre che il salto non venga effettuato. Però sarà necessario impostare come nulli i segnali in uscita dagli stadi di pipeline (nel caso di salto) utilizzando un segnale di controllo a ‘0’, mentre con l’apposita porta logica con segnale di branch andrà ad incrementare direttamente il program counter; in questo modo sarà prelevata l’istruzione corretta (sempre a patto che il salto venga eseguito).

Si può rendere più efficiente? Posso ridurre l’uso di stalli?

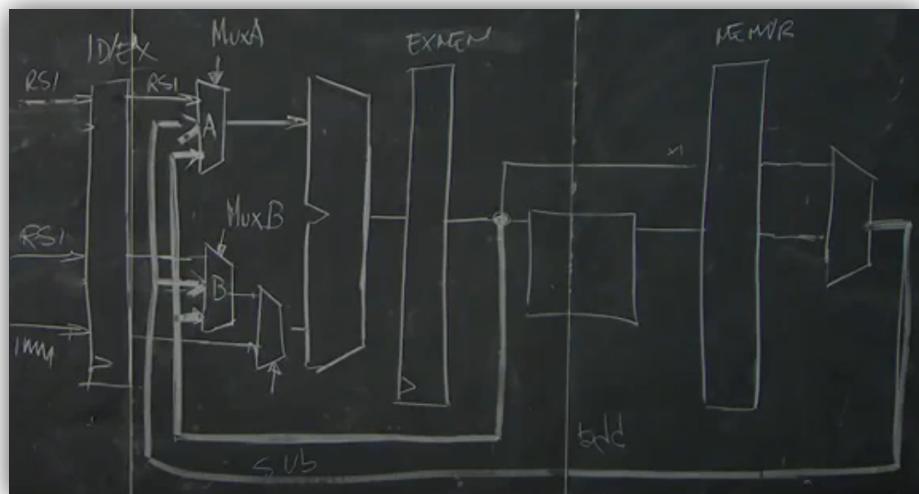
Register Forwarding: si prende l'uscita EX/MEM e si rimanda in ingresso allo stesso stadio (indicato come EX/EX o ID/EX), che visto dal punto di vista di due pipeline, è come se venisse “passato”, mentre in realtà viene messo all’indietro.



Esiste un altro tipo di *register forwarding*, il precedente era quello di tipo *EX/EX*, mentre di seguito è descritto quello *MEM/EX*; due tipologie che condividono il funzionamento ma con diverse “destinazioni”.

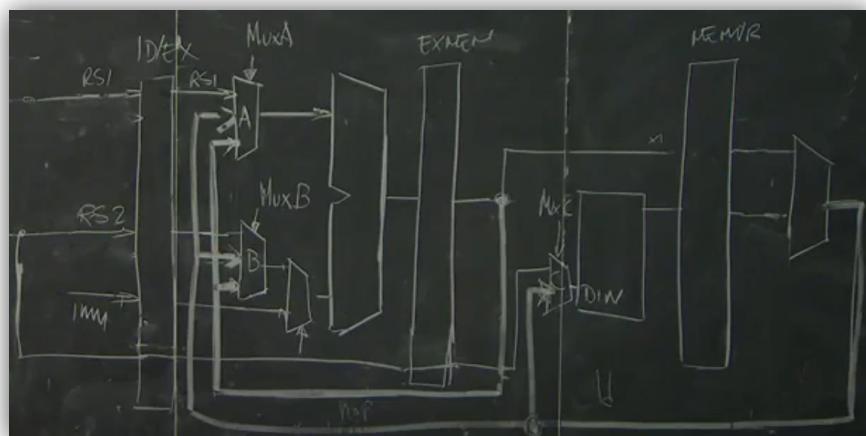
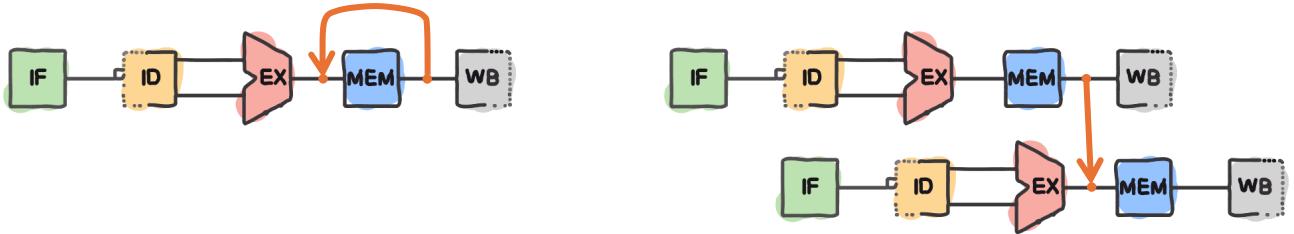


Queste ottimizzazioni hanno un risvolto pratico nell’architettura, infatti si dovranno aggiungere collegamenti e porte logiche necessarie per realizzare quanto precedentemente descritto, di seguito un modello semplificato della nuova architettura.



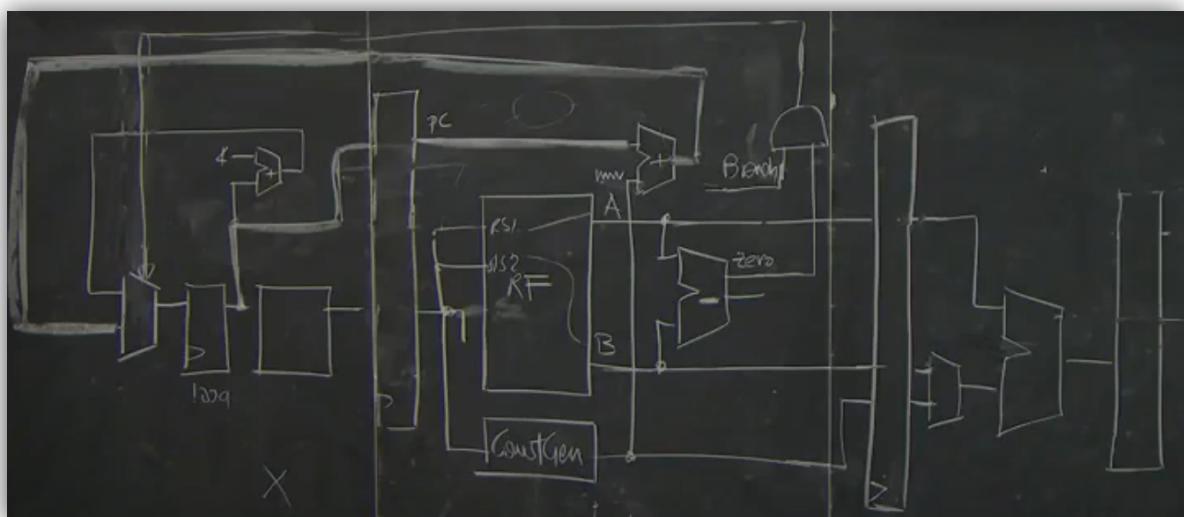
¹⁵ I tre punti indicano la presenza di un’altra istruzione tra le due.

Ma esiste un'ultima tipologia di forwarding, ed è quella da MEM/MEM (oppure MEM/WB → EX/MEM), di seguito il modello e l'architettura finale completa di tutti i forwarding necessari; è utilizzato in particolar modo per le situazioni dove si hanno due istruzioni *lw* e *sw* in sequenza (conflittuale).



Attenzione: gli esami verteranno sull'identificazione di stalli o sull'uso di NOP, e sul comprendere come le risposte cambierebbero con una pipeline ottimizzata con forwarding. Nella parte teorica, potrebbe essere richiesto di disegnare i percorsi di forwarding nella pipeline e di identificare quali stadi sono coinvolti.

A questo punto, l'architettura del RISC-V può essere ulteriormente semplificata, per cui il nuovo ed ultimo schema sarà il seguente.



Architettura – Cache e gerarchie di memoria

L'operazione di accesso alla memoria è tra le operazioni più gravose per il sistema, poiché si dovrà uscire dalla zona del microprocessore ed entrare (utilizzando i BUS) nella memoria. Tale compito rallenta il sistema non solo per tutte le operazioni necessarie all'accesso, ma anche per la capacità massima del collegamento utilizzato; anche se molto elevata, comunque introduce un ritardo e relativi disturbi dovuti a interferenze esterne.



Osservazione: con le tecnologie attuali, solitamente i bus non sono più lunghi di 50nm.

Static RAM (S-RAM): più grosse, veloci e costose, dove un bit viene memorizzato in una struttura simile ad un registro del processore.

Dynamic RAM (D-RAM): funzionano essenzialmente con un MOSFET e la sua capacità parassita che riescono a mantenere il dato per 10 – 20ms prima di un refresh; è utilizzata per le memorie RAM (memoria centrale).

Per risolvere i problemi di lentezza dati dalla DRAM, si preferisce inserire una memoria SRAM (piccola) all'interno del processore stesso, così da limitare i ritardi di accesso alla memoria centrale del sistema.

Cache L0: piccola SRAM di livello 0 interna al processore, collegata ad esso con un piccolo BUS (molto veloce); poiché il processore non ha mai bisogno di accedere a tutte le informazioni contenute nella memoria centrale, quelle che saranno utilizzate nell'immediato sono caricate nella L0.¹⁶

Con quale criterio il sistema decide che istruzioni caricare in L0?

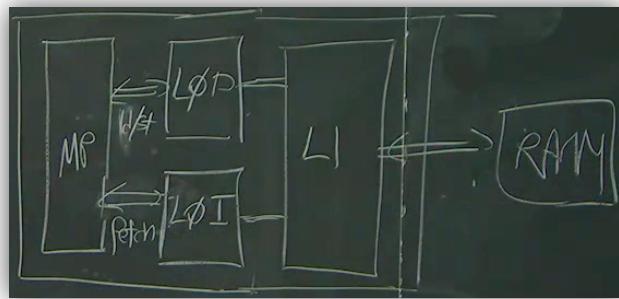
Principio di località spaziale: se in un certo momento t si accede ad A , allora in $t + 1$ (probabilmente) si accederà vicino ad A ; per questo motivo si caricheranno le zone vicine ad A in memoria Cache, per renderle subito disponibili al processore.

Principio di località temporale: dove se in un certo momento t si accede ad A , allora in $t + k$ (k piccolo) si accederà nuovamente ad A ; un esempio è il caso dei cicli. Per questo motivo si continuerà a tenere A in Cache.

¹⁶ Il tipo di architettura che fa uso di 1+ livelli di memoria Cache è l'architettura Stanford, e.g. cache L1/L2 ed L3 di un Intel i9 12900K.

Però un'architettura di questo tipo che sfrutta i principi di località, va' in crisi quando lo stesso programma prevede sia accesso ai dati che ad istruzioni successive (rispettivamente campo data e text), per questo motivo si utilizza l'architettura *Harvard* oppure la *SHARC* (super harvard architecture).

Harvard architecture: analoga a quella ideale utilizzata nel corso, che prevede \forall livello di cache due memorie diverse, una per i "dati" (L0-D) ed una per il "codice" (L0-I); nella prima vengono eseguite tutte le operazioni di load/store, mentre dalla seconda si prelevano le istruzioni in fase di *fetch*.¹⁷



Sorge ora un problema, dovuto al fatto che il programma viene scritto con riferimenti agli indirizzi di memoria RAM, non agli indirizzi della Cache: è necessario utilizzare un meccanismo per tradurre questi indirizzi. A questo scopo si usano differenti tecniche o *politiche di mapping*.

Osservazione: le richieste del processore non sono fatte ad una memoria in particolare; infatti, non è a "conoscenza" della presenza o meno della cache. Pertanto, sarà dovere della L0 quelle di occuparsi di fornire al processore quanto richiesto; si può osservare come la cache abbia un funzionamento analogo a quello di un proxy server.

Cache hit: situazione in cui il dato richiesto dal processore è disponibile in memoria cache, pertanto sarà eseguito/elaborato immediatamente. Tempo necessario $\Rightarrow 1\text{clk}$.

Cache miss: situazione opposta alla precedente, dove il dato richiesto non è presente, quindi sarà richiesto dalla cache alla RAM; una volta trasferito nella L0, quest'ultima potrà restituire il dato al processore, il tempo richiesto sarà pari a quello necessario per l'accesso in memoria (nettamente maggiore del precedente).

Hit (miss) rate: solitamente si aggira attorno al 98%, cioè su 100 accessi alla cache solo 2 (statisticamente) saranno dei *miss*. Avere un dato così ottimistico è possibile proprio per il principio di località spaziale.

Per le precedenti ragioni è necessario definire una politica di sostituzione dei dati presenti in cache, per tenere l'*hit rate* il più alto possibile.

¹⁷ Per l'architettura di riferimento, si tenga presente che le informazioni sono caricate in Cache rispettando il principio di località spaziale.

Politiche di scrittura: senza caricare/scaricare in maniera continuativa la cache, i dati possono essere modificati direttamente dalla cache senza sostituire immediatamente il blocco modificato a quello “vecchio” presente in RAM.

- *Write through*, si scrive sulla cache e “contemporaneamente” la cache stessa va ad aggiornare i dati della RAM, lasciando il blocco caricato in L0.
- *Write back*, ogni volta che una riga viene modificata, viene salvata la modifica in ram (politica che non ha bisogno del *dirty bit*, [vedi sotto]).

Politiche di sostituzione: si intende il criterio da utilizzare per sostituire le righe presenti in cache, ne esistono differenti tipologie.

- *Random*, la più semplice, la sostituzione avviene in maniera randomica (può essere realizzata impiegando dei flip-flop)
- *FIFO (First In First Out)*, il primo blocco caricato sarà anche il primo ad uscire, sicuramente migliore del precedente ma il blocco caricato per primo potrebbe ancora essere utilizzato
- *LRU (Least Recently Used)*, in questo caso viene sostituito il blocco che non viene utilizzato da più tempo; in un certo senso riprende quella che è l’idea di base del principio di località temporale

Mapping diretto: vede la cache come un elenco di linee (*linee/blocchi di cache*), ognuna delle quali ha una dimensione detta *line size* nell’ordine dei Byte (e.g. 128Byte per linea). Risulta essere la politica di mapping (traduzione degli indirizzi) più semplice, dove il numero della linea in cache è pari al numero della linea in RAM modulo il numero di linee; in altre parole, è come se la RAM risultasse suddivisa in blocchi con numero di linee uguale a quello della cache, dove si avrà una corrispondenza – *de facto* – linea per linea.

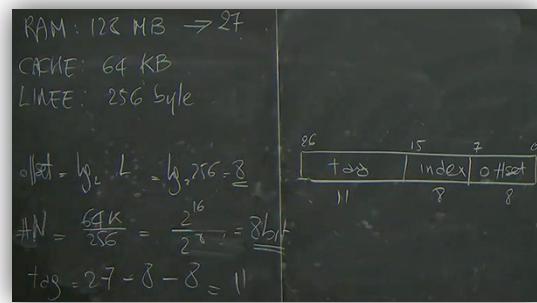
- $cache\ size = n_{line} \cdot size_{line}$ [Byte]
- Una volta scelto il “blocco” da trasferire in cache, non si dovranno compiere altre operazioni perché la corrispondenza sarà 1:1
- È necessario indicare anche quale “blocco” è stato usato, l’identificatore è detto *index*¹⁸
- Per i calcoli usare le seguenti formule/convenzioni, $L = \dim(linea)$ $S = \dim(cache)$

$$N = \frac{S}{L} = \text{numero di linee} \quad A = \text{indirizzo}_{RAM} \quad tag = \left\lfloor \frac{A}{S} \right\rfloor \quad index = \left\lfloor \frac{A}{L} \right\rfloor \% N$$

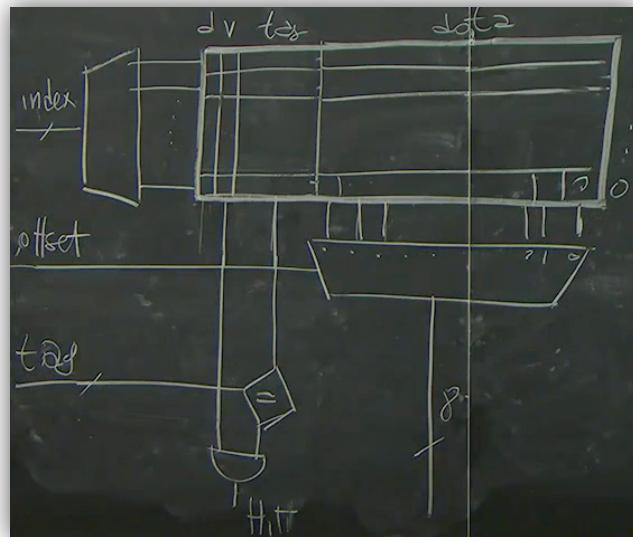
$$offset = A \% L$$

¹⁸ Sufficientemente grande per riuscire ad indicizzare tutte le linee della cache.

- È presente anche un campo da un bit, detto *dirty*, che indica se il dato è stato modificato oppure no ('1' o '0')
- Un ulteriore campo è quello del bit *valid*, che specifica se una riga è utilizzabile, oppure no
- Di seguito un esempio di risoluzione di esercizio di calcolo dell'indirizzamento diretto in cache



- Invece l'architettura di una cache che utilizza l'indirizzamento diretto, per semplicità detta *cache diretta*, è la seguente



Sarà un'apposita control unit a tenere sotto controllo il bit di *hit*, così da utilizzare o meno il dato in uscita dal decoder dell'offset; in presenza di *miss* saranno inserite centinaia di *NOP* in attesa che i nuovi dati siano caricati in cache, per poi essere restituiti (a quel punto il bit di *hit* sarà 1).

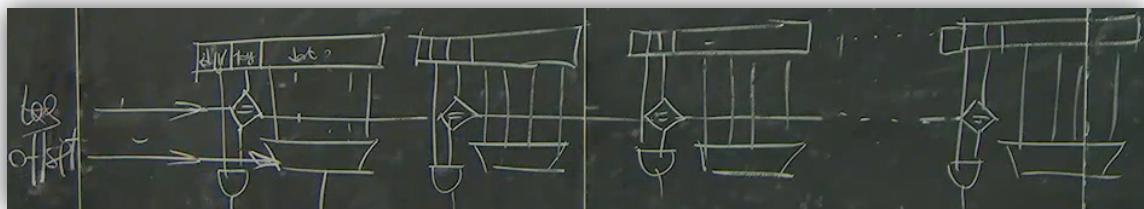
Osservazione: il miss in lettura comporta l'esecuzione di operazioni di write-back e caricamento, mentre il miss in scrittura avrà write-back, caricamento e aggiornamento. Nel secondo caso il dirty bit sarà già impostato a '1'.

Cache set-associativa a 2 vie: (o *Mapping set-ass...*) per conferire maggiore flessibilità al sistema, è possibile affiancare un ulteriore banco di memoria cache a quella già presente (identica), dove una volta scelto l'index si vanno a selezionare *due* linee di memoria; in caso di doppio miss (entrambe le linee vuote) è possibile scegliere “dove” caricare la parola cercata. Questo permette di utilizzare una memoria se l'altra è piena, minimizzando il numero di miss.

- Un esempio di risoluzione d'esercizio è il seguente

$$\begin{aligned}
 \text{RAM} &= 4GB \\
 \text{CACHE SIZE} &= 128K \\
 \text{LINEA (Byte)} &= 64 \text{ byte} \\
 \text{VIE} &= 2 \\
 \text{RAM: } &\lg_2 4G = 32 \\
 \text{offset: } &\lg_2 64 = 6 \\
 \text{Index: } &= \frac{128KB}{4GB \cdot 2W} = \frac{2^{17}}{2^6 \cdot 2^3} = 10 \text{ bit} \\
 \text{tag: } &= 32 - 10 - 6 = 16
 \end{aligned}$$

Cache completamente associativa: (o *mapping compl...*) dove si ha la possibilità indirizzare su singola linea tutte le parole (si intendono le *word*), in questo modo non è più necessario il campo index e avrà la seguente struttura.¹⁹



- Un esempio di risoluzione è il seguente

$$\begin{aligned}
 \text{RAM} &= 4GB \\
 \text{CACHE} &= 32KB \\
 \text{LINEA} &= 512B \\
 \text{Cache completamente associativa} \\
 \text{RAM} \rightarrow 4G &\rightarrow 32bit \\
 \text{offset: } &\lg_2 512 = 9 \\
 \text{tag: } &= 37 - 9 = 28
 \end{aligned}$$

Esercizi: come precedentemente illustrato riguarderanno sia il descrivere la struttura dell'indirizzo data la natura della cache, sia il determinare le prestazioni del sistema, dati i tempi di accesso, bit rate e hit/miss rate, nel caso in cui si utilizzi la cache oppure no.

¹⁹ L'indirizzo sarà composto da solamente due campi, tag(23) e offset(9).

Noto il comportamento della cache e il modo in cui si opera durante l'operazione di mapping, di seguito i principali esercizi (tratti dai temi d'esame) come la verifica del miglioramento delle prestazioni.²⁰

Osservazione: la modalità d'accesso alla memoria di tipo *burst* si può utilizzare solo in presenza di memoria cache (se presente non c'è bisogno di fare senza burst e poi con burst, se c'è la cache la usa).

Caso privo di memoria cache: data una RAM con le seguenti caratteristiche

- $size = 4GB$
- $linea = 64 \text{ bit}$
- $T_{NORMAL} = 100ns$
- $T_{BURST} = 150ns + n \cdot 60ns, \quad n \rightarrow n^{\circ} accessi$
- $T_{AVG_{access}} = T_{normal} \quad (no \ cache)$

Caso con indirizzamento diretto: nel caso in cui si usasse una memoria cache con le seguenti caratteristiche e usando i precedenti dati per la RAM.

- $size = 32KB$
- $linea = 128 B$
- $T_{HIT} = 2ns$
- $R_{HIT} = 98\%$
- $T_{MISS} = \frac{linea}{\log_2 linea/accesso} \cdot T_{NORMAL} + T_{HIT} = \frac{128B}{8B/accesso} \cdot 100ns + 2ns = 1602ns$
- $T_{CACHE} = T_{HIT} \cdot R_{HIT} + T_{MISS} \cdot (1 - R_{HIT})$
- $T_{AVG_{cache}} = R_{HIT} \cdot T_{HIT} + R_{MISS} \cdot T_{MISS} = (0.98)2ns + (0.02)1602ns = 34ns$
- Considerazioni: prestazioni triplicate, anche se pipeline non sarà piena.
- $T_{MISS}^{burst} = T_{HIT} + T_{burst}^1 + \left(\frac{linea_{cache}}{linea_{RAM}} - 1 \right) = 1052ns$
- $T_{AVG_{cache}}^{burst} = R_{HIT} \cdot T_{HIT} + R_{MISS} \cdot T_{MISS}^{burst} = 23ns$

Attenzione! il fatto che una cache è set-associativa non influenza il calcolo del tempo medio di accesso; piuttosto avrà, a differenza delle altre tipologie, un hit rate molto alto solitamente vicino al 100%.

²⁰ I tempi di spostamento sui BUS sono già “inclusi” nella voce *tempo di accesso alla RAM*, anche se spesso sono trascurabili poiché molto veloci.

Caso con indirizzamento set associativo: di seguito un esempio di calcolo, con specifiche di RAM e D-CACHE (cache per dati set associativa) a 4 vie.

RAM	D-CACHE 4w
<ul style="list-style-type: none"> • $size = 1GB$ • (linea) $W = 64bit$ • $T_{norm} = 100ns$ • $T_{burst}^1 = 120ns$ • $T_{burst}^2 = 40ns$ • 15% istruzioni $\frac{ld}{st}$ • $T_{clk} = 1ns$ 	<ul style="list-style-type: none"> • $tipo = 4way set - associative$ • $size = 128KB$ • (linea) $L = 128B$ • $T_{HIT} = 2ns$ • $R_{HIT} = 99.5\%$

- Svolgimento:

- Prima si calcola la struttura dell'indirizzo per la cache (se richiesto)
- Poi calcolo il tempo medio per l'accesso in D-CACHE

$$T_{D-Cache}^{avg} = (0.995) \cdot 2ns + (0.005) \cdot \left[2ns + \left(\frac{128}{8} - 1 \right) \cdot 40ns + 120ns \right] = 5.6ns$$

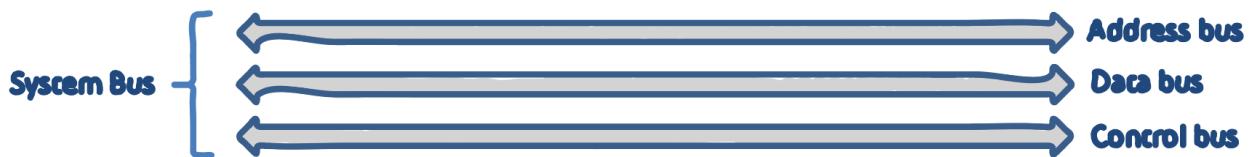
- Se presente anche l'indicazione con la percentuale di istruzioni di ld/st, allora si dovrà calcolare il tempo medio di esecuzione del programma (nanosecondi per istruzione).

$$T_{RAM}^{avg} = \frac{1 \cdot T_{Fetch} + 0.15 \cdot T_{ld,st}}{1} = 115ns, \quad T_{CACHE}^{avg} = \frac{1 \cdot T_I + 0.15 \cdot T_D}{1} = 15.44ns$$

Caso con indirizzamento full associativo: si procede come nei casi visti sopra, l'unica parte ad essere diversa (già visto nelle pagine precedenti) è quella relativa al calcolo della struttura degli indirizzi.

Architettura – Bus di sistema

Si può immaginare come tre insiemi di linee logicamente distinte, i bus di *indirizzo*, *dati* e *controllo*; ognuna di queste linee trasferisce un bit alla volta.



Bus seriali: è idealmente costituito da una sola linea bidirezionale, che trasmette un bit alla volta in maniera *half-duplex*²¹ (esistono anche in versione *full duplex*). È una soluzione hardware che presenta limitazioni, in particolare per quanto riguarda le prestazioni; inoltre sono necessarie delle trame per indicare il significato dei bit che passano sul bus.

Bus paralleli: soluzione più veloce rispetto al precedente (almeno n volte, con n che è il numero di linee), le cui linee possono essere suddivise per funzione logica svolta (addr, data e ctrl). Questa tipologia di bus è entrata in difficoltà con l'aumentare delle frequenze di funzionamento dei processori. Inoltre, per natura costruttiva sono solitamente *half-duplex*

Per questo motivo sono più si esce dalla zona del processore e si va verso le periferiche, più questi bus tenderanno ad essere seriali.

In generale, quando due componenti di un sistema comunicano attraverso il bus possono essere distinte in Master e Slave.

Sincrono: sia seriale che parallelo, dove oltre alla linea dei dati, è presente anche una linea per il Clock che sarà impartito dal Master.

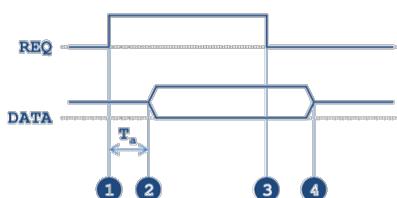
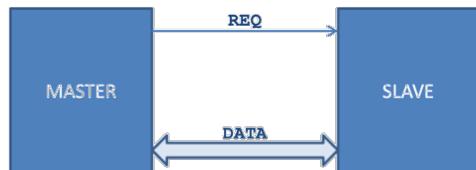
Asincrono: bus dove i clock di master e slave non coincidono, un esempio è la UART.

Tenendo presente che sarà utilizzato il seguente modello come architettura di riferimento, nelle pagine seguenti alcuni protocolli che regolano la comunicazione attraverso i bus, ciascuno con un effetto diverso sulla trasmissione (velocità, efficienza, ecc.).



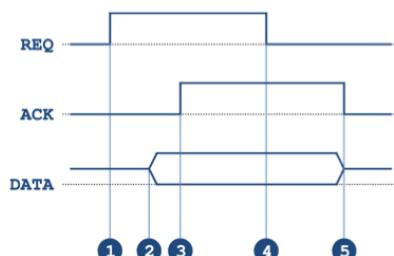
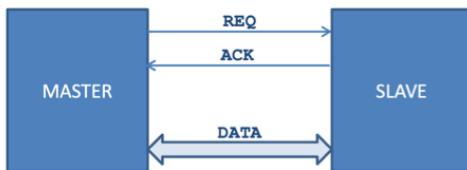
²¹ La trasmissione nelle due direzioni avviene alternativamente, poiché i dati viaggiano sulla stessa linea; può anche essere *full duplex*, ma in questo caso si avrà una linea per direzione.

Protocollo Strobe: è un protocollo di comunicazione asincrona.



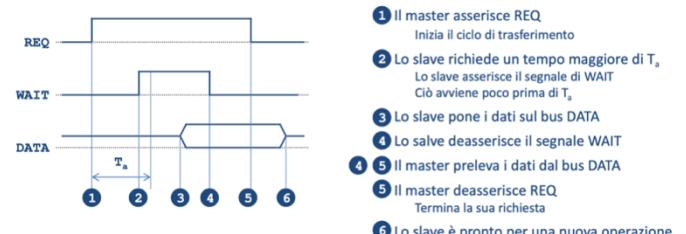
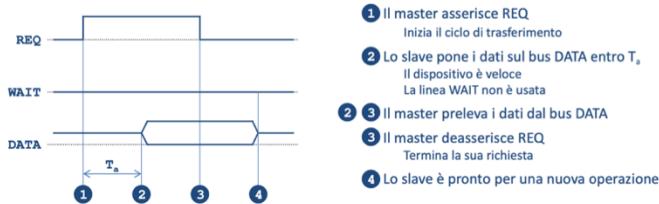
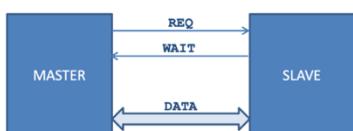
- 1 Il master asserisce REQ.
Inizia il ciclo di trasferimento
- 2 Lo slave pone i dati sul bus DATA entro T_a
- 3 Il master preleva i dati dal bus DATA
- 4 Il master deasserisce REQ.
Termina la sua richiesta
- 5 Lo slave è pronto per una nuova operazione

Protocollo di Handshake: anch'esso prevede una comunicazione asincrona.



- 1 Il master asserisce REQ.
Inizia il ciclo di trasferimento
- 2 Lo slave pone i dati sul bus DATA
- 3 Lo slave asserisce ACK
Segnala al master la disponibilità dei dati
- 4 Il master preleva i dati
- 5 Il master deasserisce REQ.
Segnala allo slave di avere ricevuto ACK
- 6 Lo slave deasserisce ACK
Lo slave è pronto per una nuova operazione

Protocollo di Strobe/Handshake: particolare protocollo che si comporta in modi differenti a seconda della velocità (trasferimento veloce a sinistra e lento a destra), di seguito i modelli esemplificativi dei due casi (con grafici dei segnali).²²

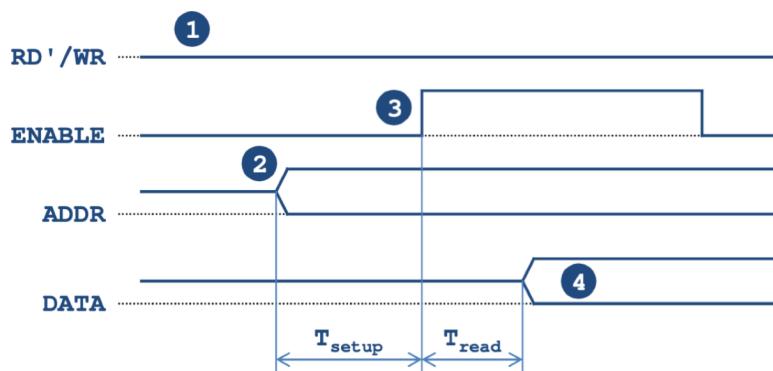


²² Un possibile esempio di utilizzo potrebbe essere quelle delle memorie cache in caso di miss.

Si possono utilizzare questi protocolli in una comunicazione sincrona?

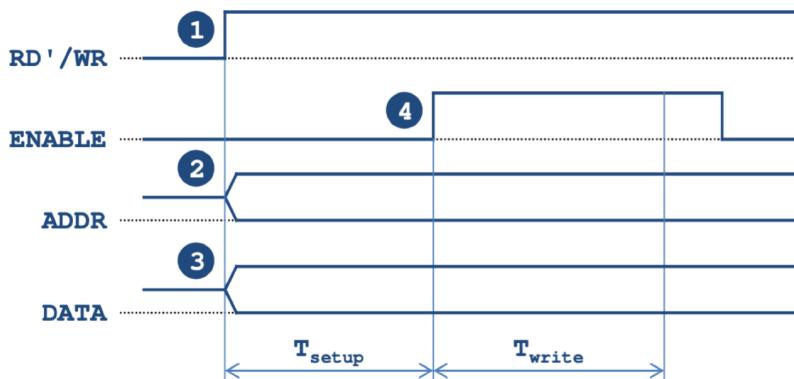
Sarà sufficiente aggiungere la linea dedicata al trasporto del segnale di clock, dove si avrà la variazione del segnale solo in presenza del fronte di clock (solitamente è usata la versione sincrona dello *strobe/handshake*): il master si sincronizza sul fronte di salita, mentre lo slave sul fronte di discesa del clock.

Ciclo di lettura :



- ① Il processore asserisce il segnale RD'
Inizia il ciclo di lettura
- ② Il processore pone l'indirizzo sul bus indirizzi ADDR
- ③ Il processore asserisce il segnale ENABLE
Almeno un tempo T_{setup} dopo che l'indirizzo è stabile
La periferica o la memoria inizia l'operazione di lettura
I dati sono disponibili sul bus dati non prima di un tempo T_{read}
- ④ Il processore preleva i dati dal bus dati DATA

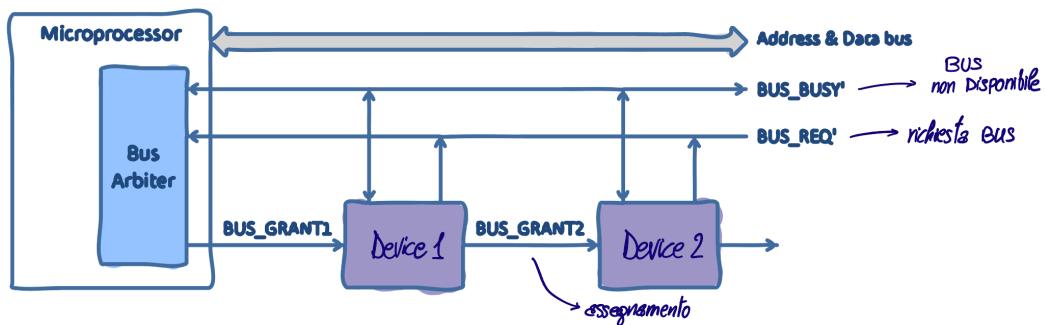
Ciclo di scrittura :



- ① Il processore asserisce il segnale WR
Inizia il ciclo di scrittura
- ② Il processore pone l'indirizzo sul bus indirizzi ADDR
- ③ Il processore pone sul bus DATA dati il dato da scrivere
- ④ Il processore asserisce il segnale ENABLE
Almeno un tempo T_{setup} dopo che l'indirizzo è stabile
La periferica o la memoria inizia l'operazione di scrittura
La scrittura sarà avvenuta non prima di un tempo T_{write}

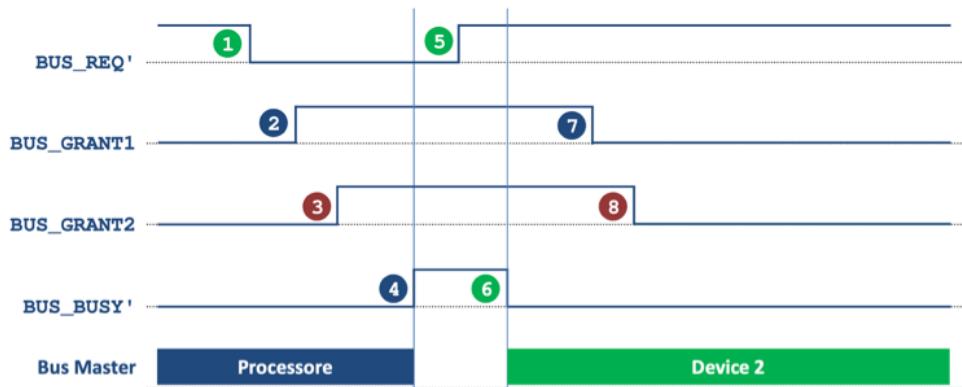
Arbitraggio: è la procedura di “negoziazione” utilizzata per decidere i ruoli *master/slave*, né esistono di due tipologie.

- *Centralizzato*, dove è un’unità della CPU a gestire l’arbitraggio.



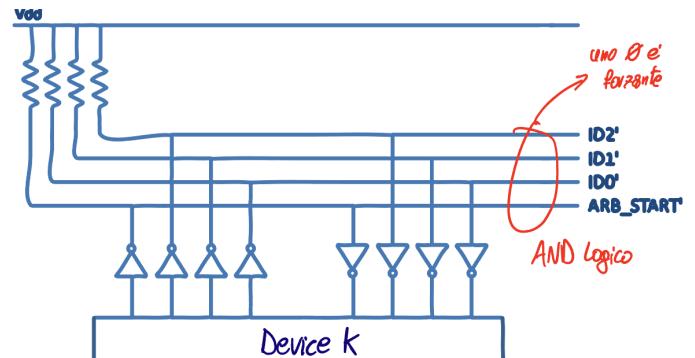
Supponendo che il processore detenga il controllo del bus e il device2 voglia richiederlo, allora la sequenza di arbitraggio prevista sarà

1. Device2 porta a 0 la linea **BUS_REQ**
2. Appena CPU vede che la linea è a 0, asserisce segnale di **GRANT** sulla linea di **BUS_GRANTED1** che viene propagato (essendo in *daisy chain*)
3. I device propagano il segnale di **GRANT**; poiché device1 non ha fatto richieste lo propaga al device2, che lo riceve e non lo propaga (perché è lui ad averlo richiesto)
4. A questo punto la linea **BUS_BUSY** vale 0 se almeno un device detiene il bus, una volta che va a 1 il device2 diviene bus Master e forza nuovamente **BUS_BUSY** a 1.



- *Distribuito*, funziona con un particolare bus rigorosamente sincrono detto CAN, dove ogni dispositivo connesso ha un particolare indirizzo ad *n bit*. Quando una periferica vuole scrivere sul bus inizia a mettere i bit, se un altro comincia a trasmettere inizia la contesa: fintanto che i bit sono identici si continua, ma appena viene forzato un bit diverso, la periferica che lo ha forzato “vince” e l’altra smette di trasmettere.

- Il sistema
 - Richiede N linee di identificazione ID0, ID1, ...
 - Tutte le linee sono di tipo open collector
 - Richiede una linea di inizio arbitraggio ARB_START
 - Anche questa linea è open collector
 - Il numero di unità massimo è pari a 2^N
- Ogni unità
 - È identificata da un codice
(a volte chiamato indirizzo) ad N bit
 - Dispone di una interfaccia piuttosto complessa



Periferica: dispositivo che permette al calcolatore di interagire con l'esterno, connessa al sistema utilizzando i bus.

Architettura – Interrupt

L'interrupt è un segnale che proviene dalle periferiche e si dirige verso il processore, "interrompendo" l'esecuzione di un programma. In altre parole, la CPU eseguirà la routine di operazioni specificata non appena (risponde in un tempo relativamente breve) riceverà l'interrupt.

Eccezioni: caso particolare, sono delle interrupt "non mascherabili" generate dal processore.

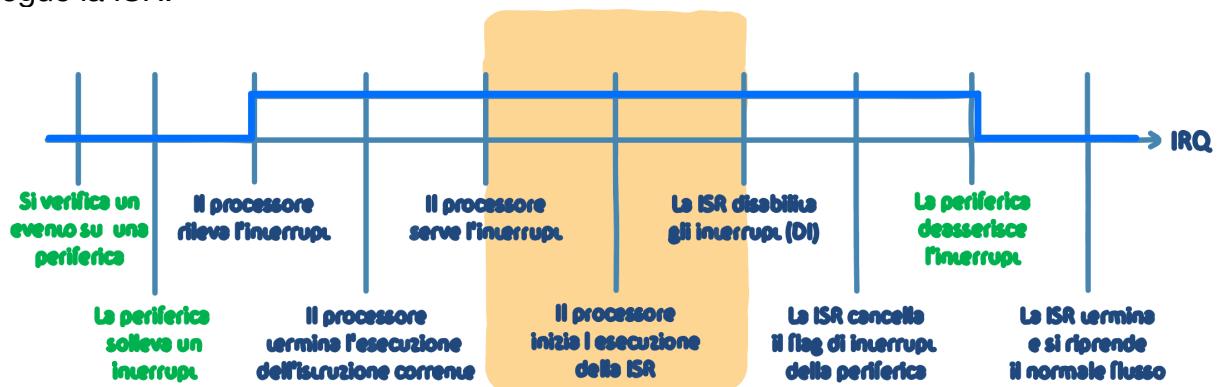
Interrupt Service Routine: insieme di operazioni/istruzioni da eseguire quando si riceve una certa interrupt, ad esempio leggere un carattere dalla tastiera (ISR) non appena si preme un tasto (Interrupt).

In questo caso come si identifica la periferica che solleva l'interrupt?

Ogni periferica è identificata da un indirizzo, che viene scritto su n linee (tante quanti i bit dell'indirizzo) e letto dal processore: se le richieste avvengono in sequenza un dispositivo vince, se invece fossero in parallelo potrebbero non essere soddisfatte le interrupt.

Nel caso in cui si ha una singola linea di IRQ si potrà eseguire una sola interrupt service routine, mentre nella situazione con linee multiple si avranno più ISR.

Ricezione Interrupt: non appena viene sollevata un'interrupt, viene inserito uno stallo in ingresso nella pipeline (fetch) ed iniziano ad entrare NOP. Dopo averne eseguite 5 allora si esegue la ISR.



Interrupt vector table: contiene gli indirizzi delle ISR, uno per ogni periferica; di seguito una mappa della memoria che illustra la gestione delle ISR e IV.

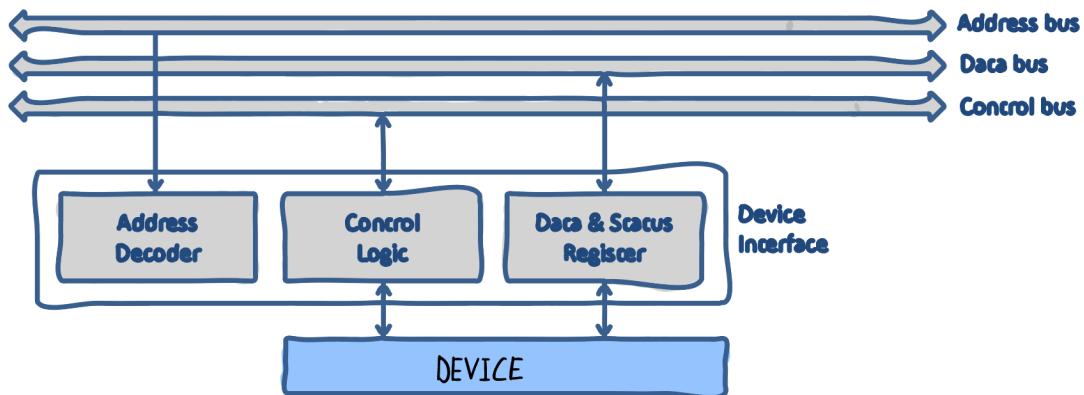
Il processore legge l'array relativo alla ISR da eseguire nell'interrupt vector e lo carica sul program counter, quindi può eseguire la ISR richiesta.

Il reset contiene l'indirizzo della prima istruzione di un programma da eseguire, istruzione contenuta nella parte di start-up.

Nota: il precedente valore del program counter viene salvato temporaneamente in un registro. Una volta terminata la ISR, tale valore sarà ricaricato nel program counter.

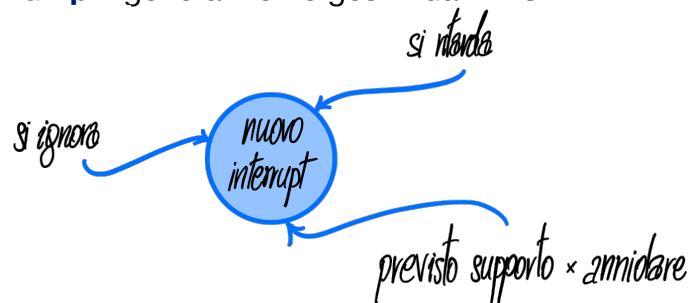
Osservazione: alcuni processori hanno 64 registri, 32 di general purpose (come quelli utilizzati nella parte di assembly del RISC-V) e 32 che sono completamente dedicati alla gestione degli interrupt, garantendone di fatto il supporto hardware.

Ogni periferica è dotata di un interfaccia in grado di comunicare con i bus di sistema ed ha registri di dati, controllo e di stato.



Tecnica di accesso ai registri delle periferiche: viene utilizzato il dispositivo di address decoder per identificare l'indirizzo presente sul bus, mentre il chip enable è utilizzato per abilitare/disabilitare l'accesso ad una periferica.

Gestione interrupt multipli: generalmente gestiti da NVIC.



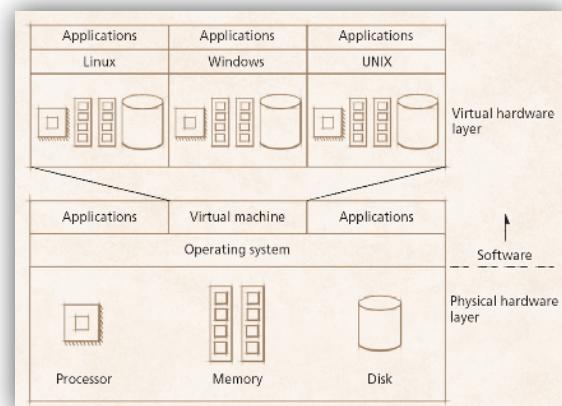
Sistema Operativo – Kernel

Il “Kernel” è il nucleo del sistema operativo, contiene tutti o parte dei componenti dell’OS, raggruppandone le funzioni centrali. In generale si occupa di *virtualizzare i processi*; nei paragrafi seguenti sarà descritto a livello sia generale, sia specifico per il corso, facendo riferimento a quello dei sistemi *linux*.

Sistema operativo: è uno strato di software che separa le applicazioni dall’hardware che utilizzano, opera come un gestore di risorse fisiche e virtuali. È utilizzato nei contesti dei sistemi *server/server farm*, *embedded systems* (e.g. *IoT*), *real-time systems* e nelle *virtual machines*²³.

Componenti OS: generalmente ogni sistema operativo ha i seguenti componenti “di base”.

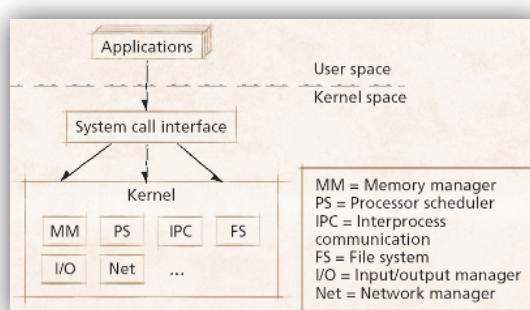
- *Scheduler*
- *Gestore della memoria*
- *Gestore I/O*
- *Gestore Inter-Process Communication*
- *Gestore del File System*
- *Shell* (interazione con utente)



Nota: la figura è una rappresentazione schematica del funzionamento alla base delle VMs.

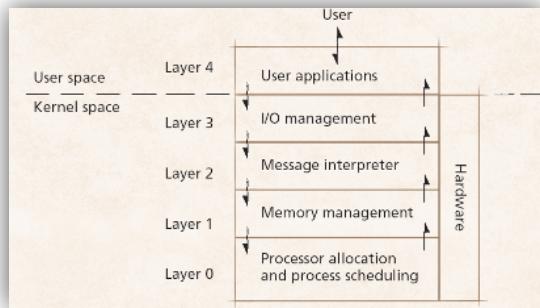
Architettura OS: la definizione di un architettura facilità l’organizzazione delle sue componenti e assegna loro i privilegi di esecuzione; esistono quattro tipologie di architetture.

- *Monolitica*, tutti i componenti sono contenuti nel kernel e comunicano direttamente; porta più efficienza a discapito della sicurezza, la mancata separazione rende difficile individuare errori

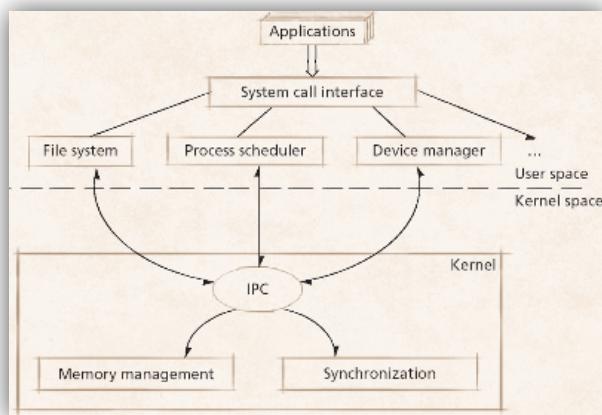


²³ Sono l’astrazione software di un computer fisico, dove il sistema operativo è in esecuzione come processo del sistema operativo nativo, al quale sono assegnate delle risorse virtuali.

- A *livelli*, raggruppa componenti con funzioni simili in un livello/strato, ognuno dei quali può comunicare solo con il livello superiore o inferiore; il tempo richiesto per attraversare più livelli porta ad un peggioramento delle prestazioni rispetto la soluzione monolitica



- *Micro-kernel*, fornisce un insieme ristretto di servizi per mantenere il kernel il più piccolo e scalabile possibile; richiede più comunicazione tra i moduli, si ha un peggioramento delle prestazioni.



- *Distribuita*, su diverse macchine fisiche

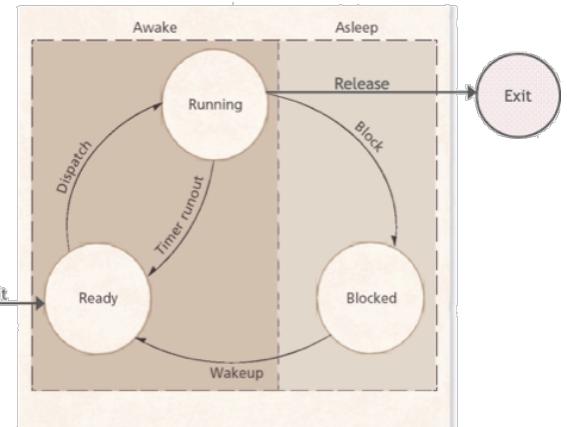
Cosa comporta la “virtualizzazione dei processi” realizzata dal Kernel?

Gestione degli interrupt: gli *interrupt* sono particolari segnali utilizzati al livello più basse dell'SO, gestito dal processore a livello hardware (in assenza di SO gestito direttamente). È sia a carico del kernel che dei driver delle periferiche. In generale la routine di gestione prevede, una volta ricevuto interrupt, di consultare una tabella (IVT, *interrupt vector table*) in cui viene indicata quale funzione eseguire per ciascun interrupt.

Routine di gestione | riconoscimento interrupt → esecuzione funzione richiesta

Operazioni: il kernel può svolgere le seguenti operazioni, dove lo stato di un processo è impostato dal sistema operativo e le transizioni da uno stato all'altro sono regolate da un modello FSM²⁴.

- *Creazione processo*, OS alloca memoria per il processo e copia codice eseguibile nella zona allocata
- *Esecuzione processo*, scheduler sceglie e mette in esecuzione un processo “ready”
- *Richiesta servizio*, processo richiede servizio all’SO con chiamata di sistema, che esegue la funzione per conto (e nel contesto) del processo
- *Sospensione su evento*, servizio richiesto deve attendere evento per esecuzione, quindi SO pone il processo in attesa e (scheduler) sceglie processo da eseguire
- *Sospensione per timeout*, SO sospende processo e lo pone in ready, quindi scheduler sceglie nuovo processo



Le informazioni sui processi sono contenute in apposite strutture dati, modellizzabili come delle struct in C; mentre gli stati come un enumerazione, di seguito un esempio di codice.

```

typedef struct ProcessDescriptor {
    ProcessStatus Status; // Status of the process
    char* StackPtr; // Stack pointer
    char* BasePtr; // Process base address
    int Event; // Event number on which process is waiting
    int Files[MAX_FILES] //Open files table
};

typedef enum ProcessStatus {
    NEW,
    READY,
    RUNNING,
    BLOCKED,
    TERMINATED
} ProcessStatus;
  
```

Osservazione: il kernel deve quindi disporre di una tabella contenente tutti i descrittori di processo ed una variabile che indichi il processo corrente, cioè l’indice del descrittore.

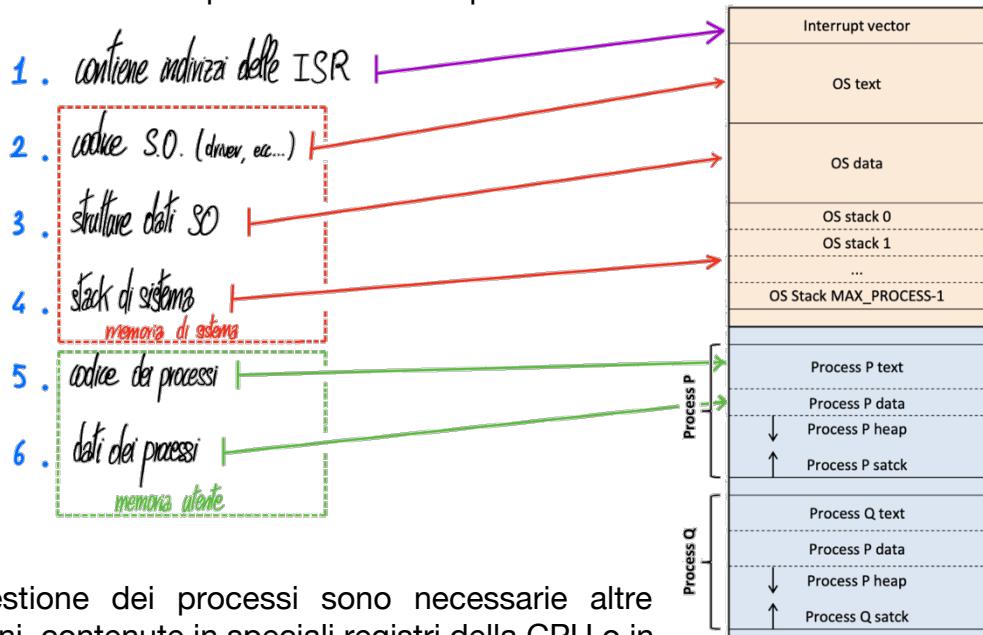
²⁴ FSM è l’acronimo di *Finished States Machine* o macchina a stati finiti; l’argomento è stato descritto ampiamente nel corso di Algoritmi e Principi dell’Informatica.

Servizi di sistema per processi: funzioni dell'SO che manipolano la tabella dei processi, tra cui le seguenti.

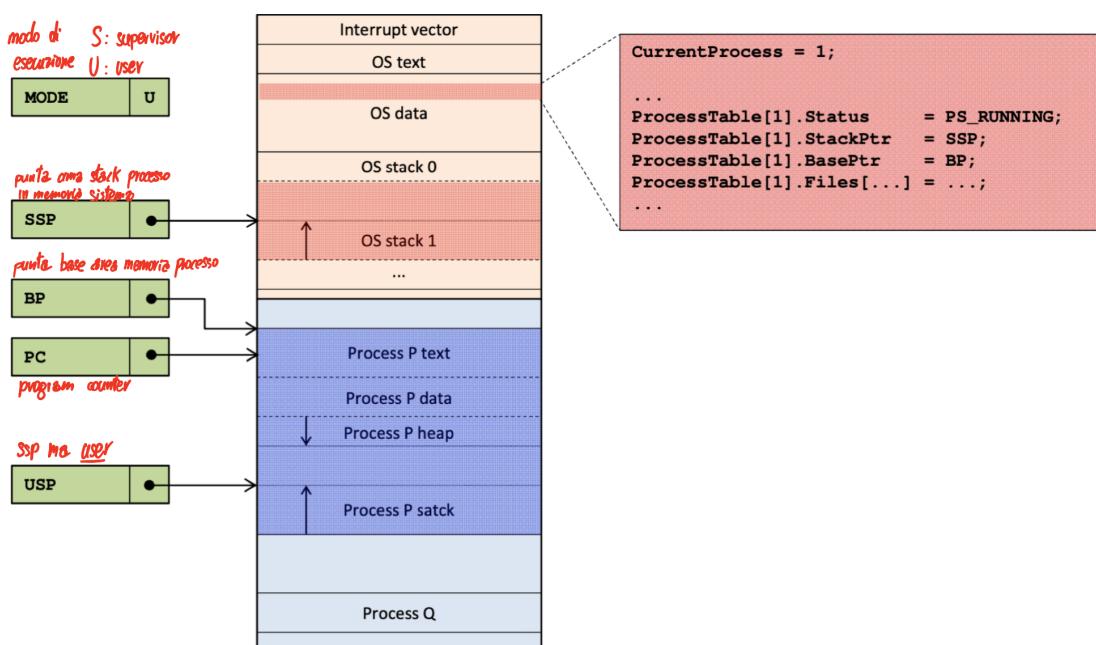
- Sospensione di un processo in attesa di un evento
- Cambio di contesto o context switch
- Ripresa di un processo sospeso su evento
- Sospensione di un processo per timeout

```
void SleepOn(int event);
void Change();
void WakeUp(int event);
void Preempt();
```

Organizzazione della memoria: in questo caso si fa riferimento all'astrazione virtuale della memoria fisica ad opera del sistema operativo.



Per la gestione dei processi sono necessarie altre informazioni, contenute in speciali registri della CPU o in locazioni nella zona di memoria dell'SO. In particolare, saranno necessarie le seguenti informazioni:



ELF: *executable and linkable format*, nei sistemi Linux è il prodotto della compilazione di un programma, è suddiviso in sezioni dove le principali sono [.text], [.data] e [.bss].²⁵

Creazione di un processo: può essere suddivisa in fasi, dove le operazioni sono eseguite dall'OS.

1. SO crea una nuova entry nella *process table* e pone lo stato a *New*
2. Alloca stack di sistema ed inizializza puntatore nella *process table*
3. Alloca la memoria nella sezione User su indicazioni dell'ELF
4. Imposta il *base pointer* all'inizio della memoria del processo
5. Copia TEXT dell'ELF nella nuova allocazione
6. Copia DATA dell'ELF nella nuova allocazione
7. Alloca memoria per BSS inizializzata a 0
8. Inizializza lo USP alla base dello *stack*
9. Inizializza tutti i campi rimanenti del *process descriptor*

Esecuzione di un processo: può avvenire su richiesta di un processo o dell'utente; per essere eseguito deve essere necessariamente nello stato di *ready* (al momento della scelta dello *scheduler*).

1. Scheduler di lungo periodo pone stato a *READY* e imposta PC
2. Scheduler pone lo stato in *RUNNING*

Richiesta servizio di sistema: a livello di codice avviene mediante chiamate a funzione di una libreria.

1. Processo richiede servizio mediante istruzione SVC all'indirizzo *BP + Y*
2. Sistema entra in *S mode*, salva indirizzo e modo di ritorno su *system stack*
3. PC punta al codice del gestore di servizi
4. Il sistema esegue servizio in *S* nel contesto del processo
5. Terminato il servizio ⇒ sistema ripristina PC e modo salvati

²⁵ Questa parte è stata già descritta nel primo capitolo, di introduzione alla programmazione in C per sistemi Linux.

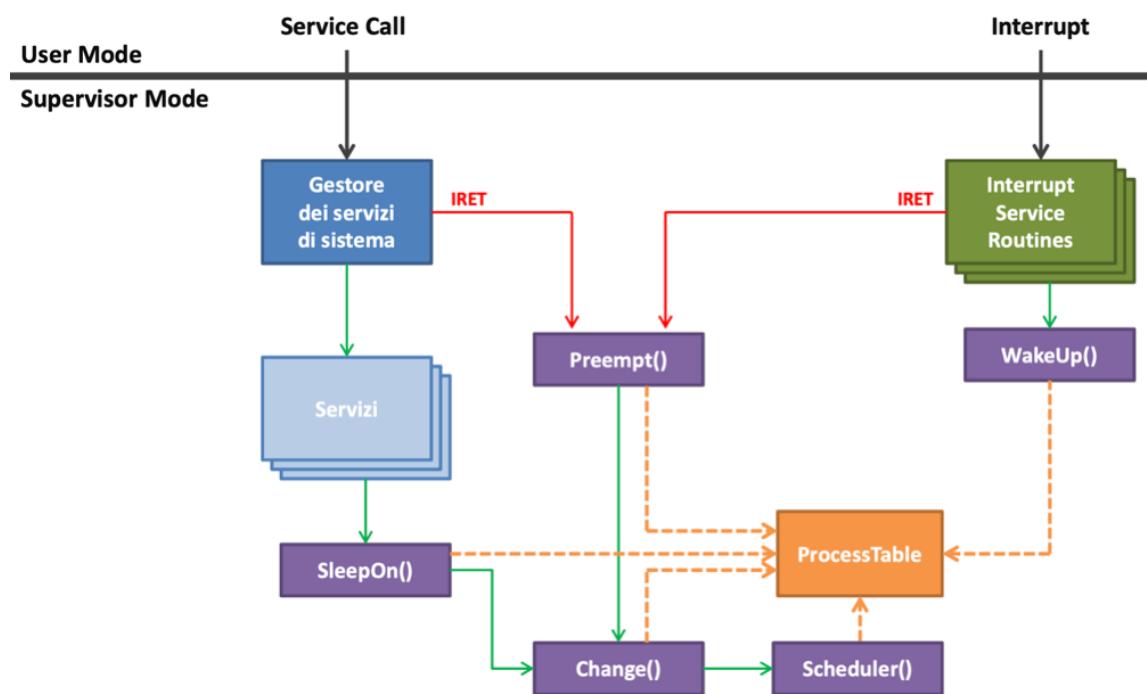
Sospensione di un processo: si verifica quando l'esecuzione di un servizio non può essere completata, solitamente per *interrupt*, *indisponibilità dei dati*, *rilascio dei lock*.²⁶

1. Durante l'esecuzione del servizio, il processo si sospende in attesa di un evento
 - a. Salva tutte le informazioni del processo (contesto)
 - b. Cambia lo stato in READY
 - c. Effettua cambio di contesto (nuovo processo in esecuzione)
2. Viene usata *Change(...)*; per cambio di contesto

Ripresa di un processo (sospeso): grazie alla funzione di *wake up* vengono risvegliati tutti i processi bloccati e viene restituito il controllo alla ISR. Dopodiché, la ISR termina e l'esecuzione continua normalmente.

Esaurimento del quanto di tempo (timeout): se processo *Q* esaurisce il suo quanto di tempo (misurato da ISR associata al *system clock*)²⁷. Periodicamente il sistema invoca la funzione *Preempt(...)* per verificare se il processo corrente è in *timeout*; in caso positivo, viene effettuato il cambio di contesto.

A questo punto è possibile definire la struttura generale del Kernel, note le sue operazioni.



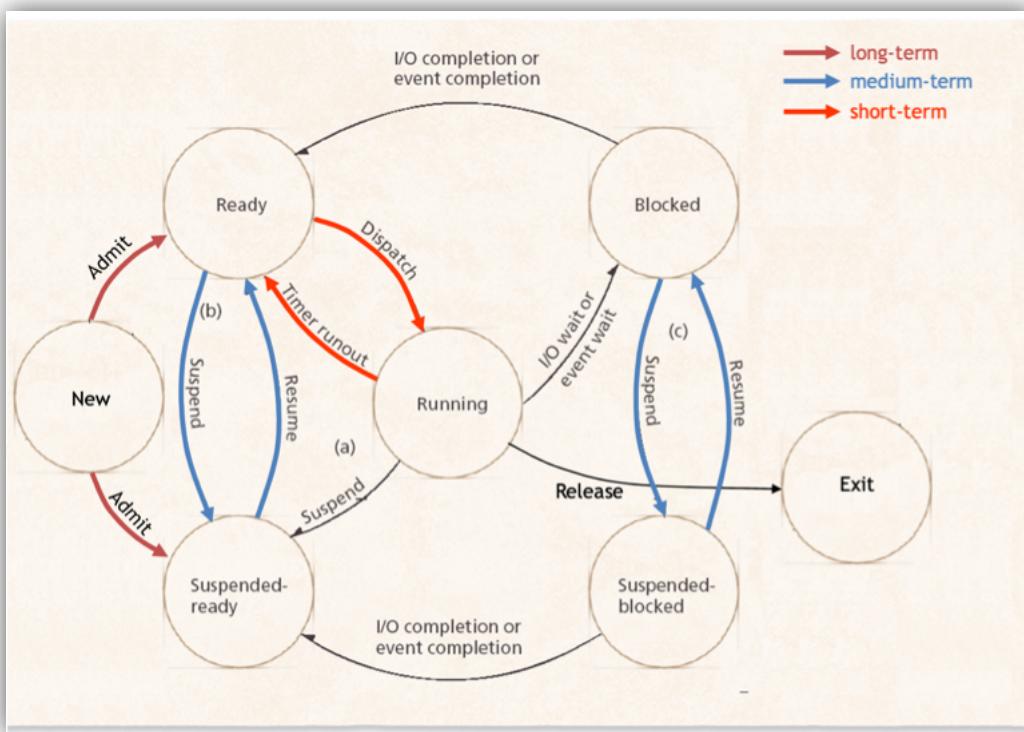
²⁶ La gestione dei processi da parte del sistema operativo è strettamente legata alla parte di programmazione di sistema.

²⁷ Un interrupt sollevato dal clock è detto *tick*; una variabile globale mantiene il tempo corrente e viene incrementata ad ogni *tick* della ISR del clock.

Sistema Operativo – Scheduling

È l'operazione di decisione su quale processo eseguire in un dato istante, né esistono differenti livelli in un sistema operativo, in particolare lo *scheduling long term*, *middle term* e *short term*. I precedenti tre si occupano rispettivamente di:

- Determinare quali processi iniziare (controlla numero di processi in un sistema)
- Determinare quali processi possono competere per il processore (reagisce a fluttuazioni di breve durata nel carico del sistema)
- Assegna CPU a processi e gestisce le priorità



28

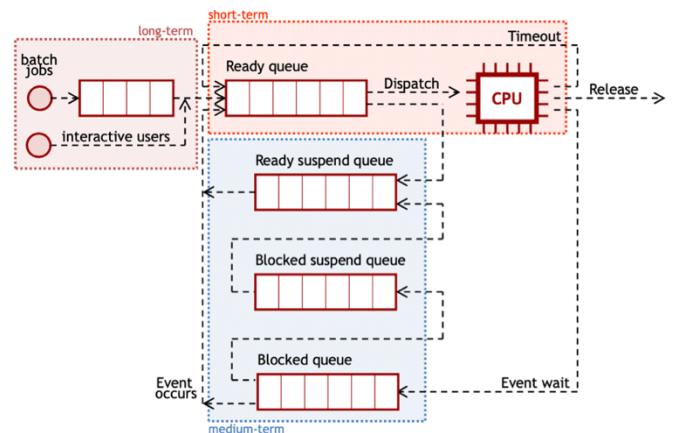
Scheduler: è il componente fisico che realizza lo scheduling utilizzando appositi algoritmi per compiere le scelte opportune.

Dispatcher: è lo scheduler di breve periodo, eseguito di frequente ed invocato al verificarsi di un evento (e.g. *interrupt*); può essere *user oriented* per minimizzare il response time, oppure *system oriented* per un uso efficiente della CPU.

²⁸ Grafo che rappresenta il ciclo di vita dei processi in relazione a quanto viene deciso dallo scheduler, separando le responsabilità per scheduling di lungo/medio/breve termine.

Politiche: sono prevalentemente di due tipi

- *Preemptive P*, consente di sospendere i processi, migliora la latenza (adatto ad ambienti interattivi, interrompibili)
- *Non preemptive NP*, dove i processi lasciano il controllo della CPU solo se terminano o lo fanno deliberatamente (non interrompibili)



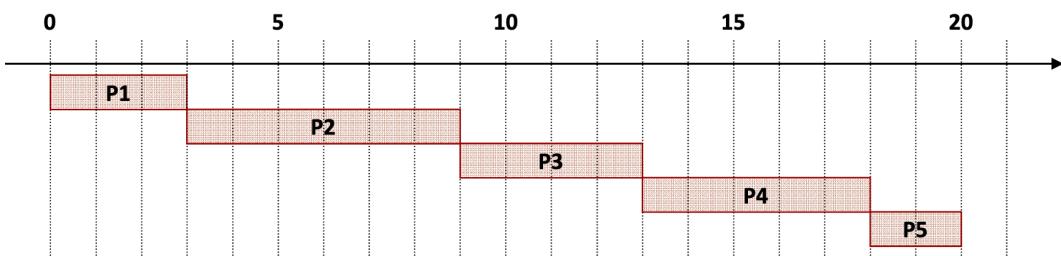
Priorità di esecuzione: viene assegnata ad un processo, e può essere

- *Statica*, assegnata alla creazione è facile da implementare, ha un basso overhead ma non permette reazioni efficienti del sistema a variazioni dell'ambiente (e.g. carico)
- *Dinamica*, reattiva ai cambiamenti, consente una buona interattività ma porta un maggior overhead

Di seguito un elenco degli algoritmi di scheduling studiati nel corso, accomunati da aspetti come *fairness*, *predicibilità* e *scalabilità*.

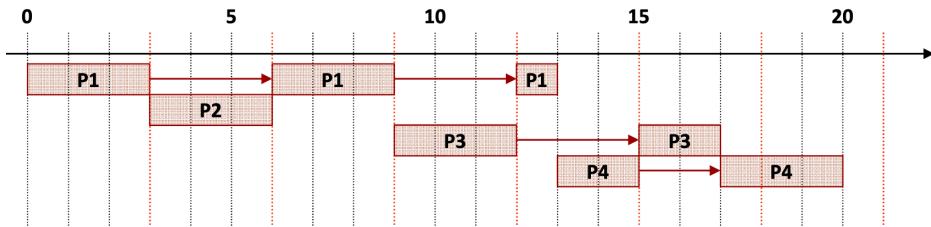
First Come First Serve FCFS (o FIFO): è l'algoritmo di scheduling più semplice, di tipo NP, dove i processi vengono serviti (dispatched) in base al tempo di arrivo. Però, potrebbe non eseguire (per molto tempo) un processo breve; inoltre, favorisce i processi CPU bound.

Processo	Arrival time	Service (run) time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2



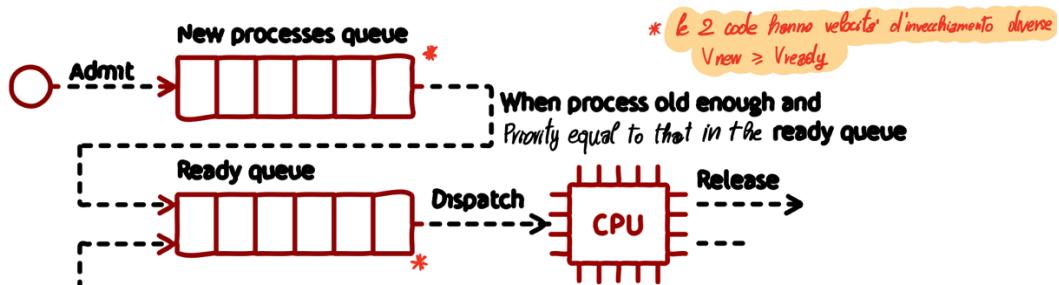
Round Robin (RR): è un algoritmo FIFO di tipo P, introduce un tempo massimo di esecuzione per ciascun processo, se il processo non termina entro il tempo limite viene rimesso in fondo alla coda.

Processo	Arrival time	Service (run) time
P1	0	7
P2	2	3
P3	5	5
P4	10	5



Osservazione: come il precedente, anche il RR, generalmente non viene scelto come algoritmo di scheduling principale.

Selfish Round Robin (SRR): variante dell'RR, dove ogni processo ha una priorità che aumenta man mano che il processo “invecchia” (senza essere eseguito), viene utilizzata una coda aggiuntiva. Di seguito una descrizione del funzionamento dell'algoritmo.



1. Un processo entra nella coda dei processi nuovi
2. Tale processo invecchia e aumenta la priorità
3. Quando la priorità è pari a quella dei processi ready
 - a. Entra nella coda dei processi ready
 - b. È ora soggetto all'algoritmo di scheduling principale (RR)

Oltre alla condizione indicata nello schema precedente (si osservino le velocità di invecchiamento), affinché l'algoritmo funzioni, i processi nella coda dei processi nuovi devono poter entrare nella coda ready per poter essere soggetti a dispatch²⁹.

²⁹ Per essere eseguiti, i processi devono obbligatoriamente trovarsi nella coda dei processi ready.

In funzione della relazione tra le velocità di invecchiamento delle due code si avrà:

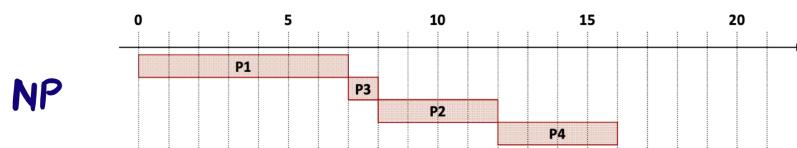
- $V_{new} > V_{ready}$ (maggiore di poco), processi resteranno in attesa per poco tempo ma non trascurabile.
- $V_{new} \gg V_{ready}$, la politica SRR degenera in RR
- $V_{new} = V_{ready}$, la politica SRR degenera in FIFO

Shortest Job First (SJF): algoritmo che assegna ad ogni processo la lunghezza del prossimo *CPU burst*, usando tali informazioni per scheduling del processo successivo; può essere sia NP (attende fine burst corrente) oppure P dove se arriva processo con tempo inferiore a quello rimasto dell'attuale, questo viene sospeso e mandato in esecuzione. Nell'ultimo caso, è detto *Shortest Remaining Time First (SRTF)*; la durata del burst successivo viene stimata come³⁰

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n, \text{ con } 0 < \alpha < 1$$

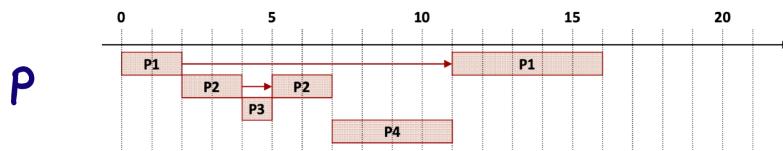
È un algoritmo che minimizza il tempo d'attesa medio di un insieme di processi, a patto che α rispetti il vincolo imposto.

Processo	Arrival time	Burst time
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Tempi di attesa: P1=0, P2=6, P3=3, P4=7

Tempo medio di attesa: $(0+6+3+7)/4=4$



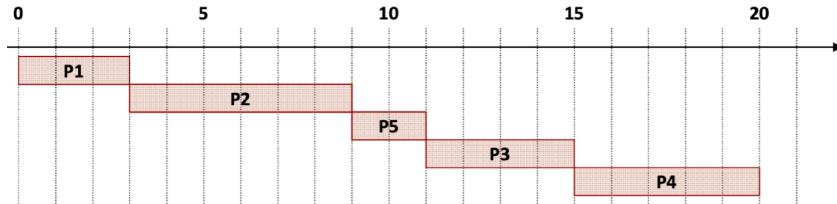
Tempi di attesa: P1=0+9, P2=0+1, P3=0, P4=2

Tempo medio di attesa: $(9+1+0+2)/4=3$

³⁰ t_n indica la durata del burst corrente; la formula usata è una relazione basata sulla media esponenziale.

Shortest Process Next (SPN): algoritmo NP dove viene scelto il processo *ready* con runtime stimato minore; i processi brevi sono serviti rapidamente, a discapito di quelli lunghi, introducendo problemi di potenziale *starvation* e diminuzione della predicitività per tali processi.

Processo	Arrival time	Service (run) time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2



Tempi di attesa: P1=0, P2=1, P3=7, P4=9, P5=1

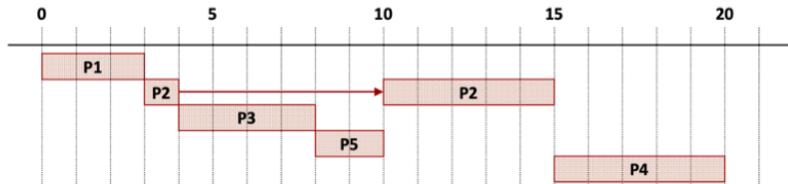
Tempo medio di attesa: $(0+1+7+9+1)/5=3.6$

Highest Response Ratio Next: è un algoritmo NP che al momento del context switch calcola la priorità di ogni processo, eseguendo il processo selezionato fino al completamento, favorendo i processi brevi e quei processi che attendono da molto.³¹

$$\text{Priorità} = \frac{T_{attesa} + T_{exec}}{T_{exec}}$$

Shortest Remaining Time (SRT): è la versione P dello Shortest Process Next, richiede un'accurata stima dei tempi di esecuzione.

Processo	Arrival time	Service (run) time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2



Tempi di attesa: P1=0, P2=1+6, P3=0, P4=9, P5=0

Tempo medio di attesa: $(0+7+0+9+0)/5=3.2$

³¹ Di fatto, risolve le principali problematiche introdotte con l'utilizzo degli algoritmi SJF/SPN.

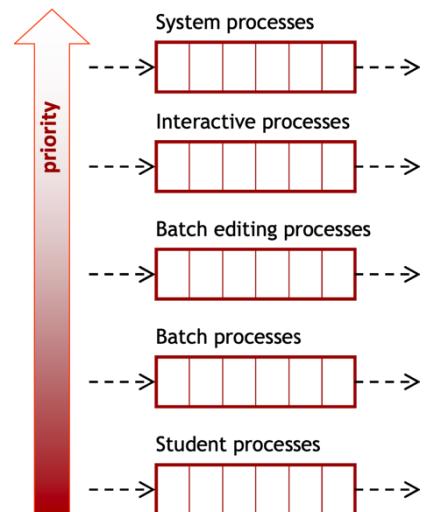
Multilevel Queues (MQ): algoritmo che prevede più code, in particolare la coda dei processi ready viene suddivisa in altre due, ognuna con il proprio algoritmo di scheduling.

- *Coda processi in foreground* (interattivi), utilizza RR
- *Coda processi in background* (batch), utilizza FCFS

Data questa caratteristica, è necessario che si effettui una sorta di scheduling anche tra le code.

In particolare, utilizzando il *fixed priority scheduling* che prevede l'esecuzione dei foreground prima e dei background poi (possibile starvation).

Oppure il *time slice scheduling* dove ogni coda ha una frazione di tempo d'esecuzione fissata. Inoltre, è facilmente generalizzabile su più livelli.



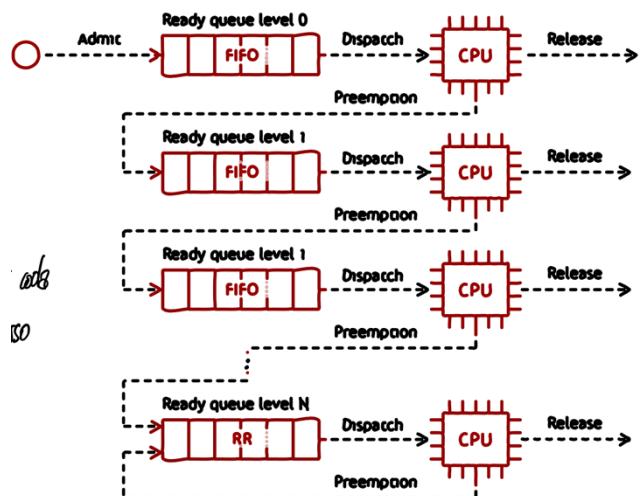
Multilevel Feedback Queues (MFQ): algoritmo che sfrutta un meccanismo di code multilivello a retroazione, il cui funzionamento è descritto di seguito.

1. I nuovi processi entrano nella coda a livello più alto e vengono eseguiti con priorità maggiore rispetto i processi nelle altre code
2. I processi più lunghi scendono nelle code a priorità minore; con processi brevi e I/O bound che mantengono priorità maggiore, mentre i processi lunghi verranno eseguiti al termine di quelli brevi e I/O.

Le politiche di scheduling delle diverse code possono essere RR o FIFO.

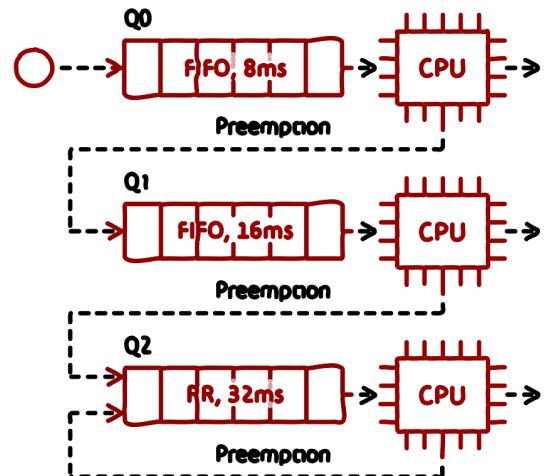
Scheduler MFQ: uno scheduler basato su code multilivello con feedback è definito da

- *numero di code*
- *algoritmi di scheduling* ∀ coda
- *criteri di definizione del livello iniziale* (di un processo)
- *criteri di upgrade* (dei processi)
- *criteri di demote* (dei processi)



Esempio: di seguito un esempio di scheduling con MFQ e relativo schema.

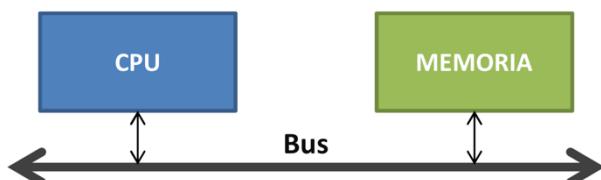
- **Code**
 - Q0: FIFO, preemptive, 8ms
 - Q1: FIFO, preemptive, 16ms
 - Q2: RR, preemptive, 32ms
- **Scheduling**
 - Nuovo processo entra in Q0
 - Riceve 8ms di CPU
 - Se non termina è interrotto e passa in Q1
 - In Q1
 - Riceve 16ms di CPU
 - Se non termina è interrotto e passa in Q2
 - In Q2
 - Viene completato in burst di 32ms



Sistema Operativo – Memoria Virtuale

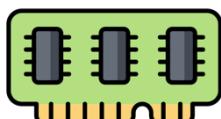
Per memoria si intende la RAM del sistema, costituita a livello fisico da un insieme di celle contigue da 1Byte. Gli accessi avvengono solitamente per gruppi di 2, 4 o 8 byte consecutivi detti *parole* (words).

Accesso allineato: (modalità d'accesso) prevede che l'indirizzo d'inizio di una parola sia un multiplo della sua dimensione.



Indirizzamento: necessario per accedere alle celle di memoria e prelevare i dati; lo spazio di indirizzamento dovrà essere sufficientemente grande per riferirsi a tutte le celle.

Solitamente i sistemi hanno una quantità di memoria inferiore rispetto quella indirizzabile



Memoria fisica: numero di Byte effettivamente (a livello fisico) disponibile sul sistema.

Memoria virtuale: numero di Byte indirizzabili da una parola di microprocessore, dove una parola di $n - bit$ può dare origine a 2^n indirizzi diversi \Rightarrow si ha uno spazio di indirizzamento di 2^n celle. Fornisce un'astrazione della memoria fisica, ma indipendente da questa), usata come modello di riferimento dal programmatore.³²

Indirizzo logico: generato dal linker a fine compilazione, sempre partendo da un indirizzo fisso (e.g. '0').

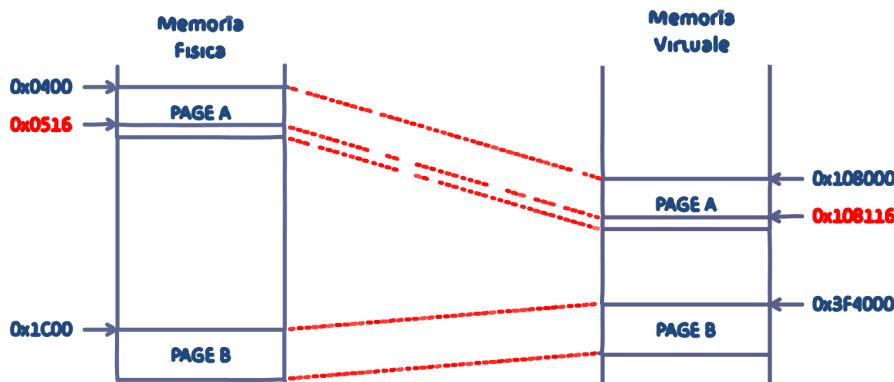
Rilocazione: processo dove il sistema operativo aggiorna gli indirizzi in base alla posizione effettiva del codice in un contesto privo di memoria virtuale; in realtà è presente anche con le memorie virtuali.

La traduzione indirizzo *logico \leftrightarrow fisico* è realizzata dalla *MMU* a livello hardware e dal gestore di memoria virtuale OS a livello software; la traduzione è anche indicata come *memory mapping*.

In breve, un programma in esecuzione non risiede – in generale – completamente nella memoria fisica del sistema; da cui se ne deduce che parti di uno stesso programma devono essere caricate/scaricate dinamicamente dalla memoria (operazione eseguita da OS).

³² Infatti, i programmi e le applicazioni fanno riferimento proprio agli indirizzi virtuali o *logici*.

Paginazione: soluzione³³ basata parti di memoria a dimensione fissa dette *pagina*, con dimensioni che vanno da 1KB fino a 64KB. Memoria fisica e virtuale sono viste come un insieme di pagine contigue, di seguito uno schema riassuntivo.



- Ad esempio, facendo riferimento alla pagina A
 - Indirizzo virtuale d'inizio: 0x108000
 - Indirizzo virtuale del dato: 0x108116
 - Indirizzo fisico d'inizio: 0x0400
 - Indirizzo fisico del dato: 0x0516

Consideriamo l'indirizzo del e scomponiamolo in due parti

- Numero di pagina (verde) e offset (azzurro)
- Considerando gli indirizzi virtuali, si ha



- Mentre nel caso di indirizzi fisici, si ha



Memory Map: per ogni pagina allocata associa il numero di pagina logica al numero di pagina fisica. Facendo riferimento al precedente esempio, si avrebbe una tabella come di seguito.

Memory Map	
VPAGE	PPAGE
0x108000	0x0420 0x01
0x3F4000	0x0FD0 0x07

Arrows point from the Virtual Page numbers (0x108000 and 0x3F4000) to their respective entries in the Memory Map table. Arrows also point from the Physical Page numbers (0x0400 and 0x1C00) to the bottom of each row in the table.

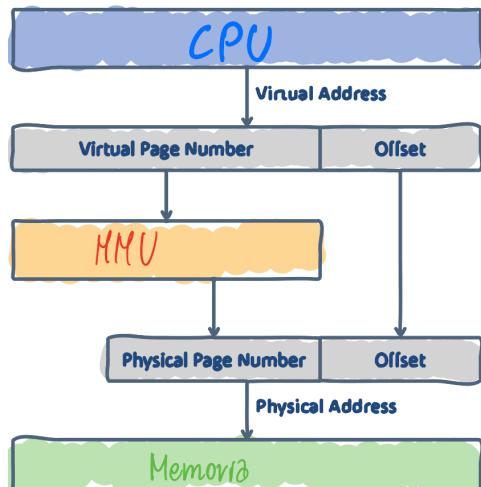
Il funzionamento di quest'operazione può essere suddiviso in 4 passi, dato a un indirizzo logico:

- Lo si spezza nei campi *VPAGE*, *OFFSET*
- Si individua nella memory map la riga corrispondente alla *VPAGE*
- Si preleva dalla memory map la corrispondente *PPAGE*
- Si combina la *PPAGE* con il campo *OFFSET* (ricavato al passo 1)

³³ Per caricare o scaricare parti di programma dalla memoria fisica.

Osservazione: questa è un'operazione dispendiosa, per tale motivo non è affidata “totalmente” al sistema operativo, si è scelto di implementare un apposita unità hardware preposta allo scopo (MMU).

Memory Management Unit (MMU): componente basato su una memoria associativa che prende in ingresso un indirizzo virtuale e produce in uscita un indirizzo fisico, a patto che sia stata configurata con una specifica mappa di memoria (*memory map*).



Ogni processo dispone della propria mappa detta *MMU Table*, che passerà alla MMU; la tabella di un processo contiene solo parte delle pagine (del processo), mentre la MMU contiene le tabelle dei processi in esecuzione.

In questo modo il processo occuperà solo una piccola parte della memoria fisica, rispetto alla dimensione della memoria virtuale.

Esempio: dati due processi P e Q, di seguito lo schema di traduzione che avviene all'interno della MMU.

Virtual Memory P (PID=1)		MMU Table			Physical Memory	
VPAGE	CONTENT	PID	VPAGE	PPAGE	PPAGE	CONTENT
0	AAAA	1	0	1	0	
1	BBBB	1	1	2	1	AAAA
2	CCCC	1	2	6	2	BBBB
3	DDDD	1	3	9	3	XXXX
...	4	
					5	YYYY
					6	CCCC
					7	
					8	
					9	DDDD
					10	ZZZZ
					11	
					12	
					13	WWWW
					...	

Virtual Memory Q (PID=3)		MMU Table			Physical Memory	
VPAGE	CONTENT	PID	VPAGE	PPAGE	PPAGE	CONTENT
0	XXXX	3	0	3	0	
1	YYYY	3	1	5	1	YYYY
2	ZZZZ	3	2	10	2	ZZZZ
3	WWWW	3	3	13	3	WWWW
...	

Struttura tabella: i sistemi Linux usano una struttura gerarchica, dove l'indirizzo logico viene scomposto

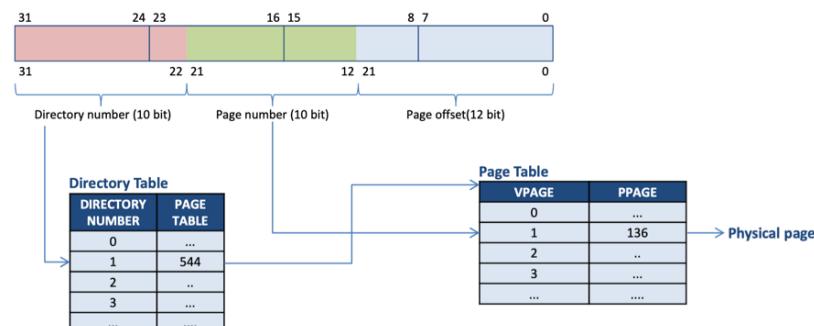


Table Miss: situazione dove viene richiesto l'accesso ad una pagina virtuale non attualmente presente in MMUT, comporta una perdita di efficienza. Per ridurre la probabilità di *miss*, è necessario che la dimensione della porzione di tabella associata al processo si avvicini il più possibile al numero R di pagine del processo residenti in memoria.

Condivisione: alcuni processi possono condividere pagine fisiche, né è un esempio quanto accade con la funzione *fork(...)*, ma dal punto di vista della MMU non cambia nulla.

Protezione: realizzata associando ad ogni pagina virtuale di un processo alcuni bit “di protezione” che definiscono le modalità d’accesso consentite; se violate, viene generato un’interrupt.³⁴

Page Fault: si ha quando viene richiesto l’accesso ad una pagina virtuale che era stata tolta dalla memoria fisica con lo *swap out* (operazione, OS sposta pagina RAM → disco), quindi il processo viene sospeso con interrupt in attesa che venga fatto *swap in* (ricaricamento pagina in RAM). Per indicare se una pagina è caricata in memoria (oppure no) è utilizzato il *valid bit*, ‘1’ se presente, ‘0’ altrimenti.



Quando si ricarica una pagina in memoria dall’area di *swap*, sarà necessario lo *swap out* di una pagina attualmente caricata³⁵; per scegliere quale si utilizzano altri due bit.

- *Access bit*, posto a ‘0’ appena si carica la pagina in memoria, ‘1’ per ogni volta che viene richiesto accesso ad essa.
- *Dirty bit*, posto a ‘0’ appena si carica la pagina in memoria, ‘1’ quando si accede in scrittura ad una parola della pagina; utilizzato per decidere se aggiornare la copia su disco della pagina al momento dello *swap out* (nel caso in cui vale ‘1’).

Con che criteri, o politiche, si sostituiscono le pagine in memoria?

Last Recently Used (LRU): sfrutta l’*access bit* per misurare l’invecchiamento → periodicamente OS pone l’*access bit* a ‘0’.

- *LRU semplice*, se *access bit* = 1 allora è stato usato nell’ultimo periodo di reset, quindi rimane in memoria
- *LRU avanzato*, prevede l’utilizzo di un contatore che sarà incrementato in tutte le pagine dopo ogni periodo se hanno ancora *access bit* = 0

³⁴ Le modalità di accesso sono essenzialmente *lettura R*, *scrittura W*, *esecuzione X*.

³⁵ La memoria fisica è piena, pertanto prima di ricaricare una pagina è necessario liberare spazio.

Working Set: di ordine k , è l'insieme delle pagine usate negli ultimi k accessi a memoria; secondo i principi di località e k sufficientemente grande, il *working set* cambia molto lentamente \Rightarrow si riduce il fenomeno di *Page Fault*.

- R è la dimensione del *working set*

Demand paging: è la sequenza di caricamento iniziale delle pagine, anche detto *lazy loading*.

Osservazione: un'alternativa consiste nel caricare subito tutte le pagine del programma.

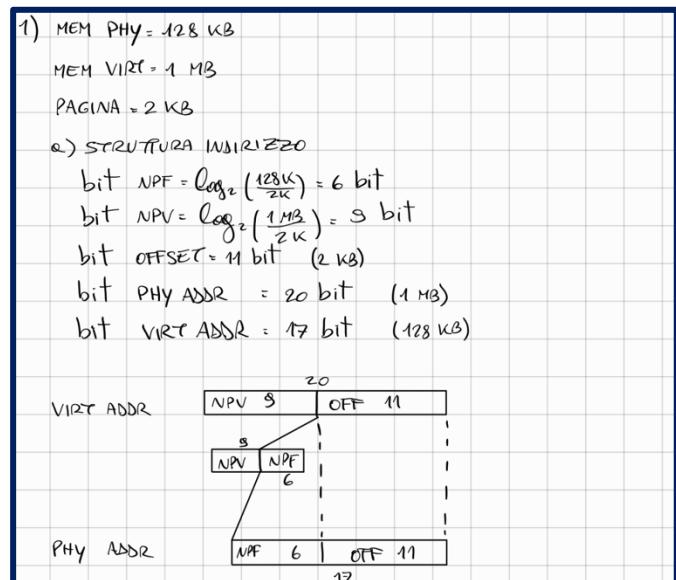
Risoluzione esercizi sulle memorie virtuali

La prima tipologia o prima parte di un esercizio riguarda il calcolo della struttura degli indirizzi fisico e virtuale.

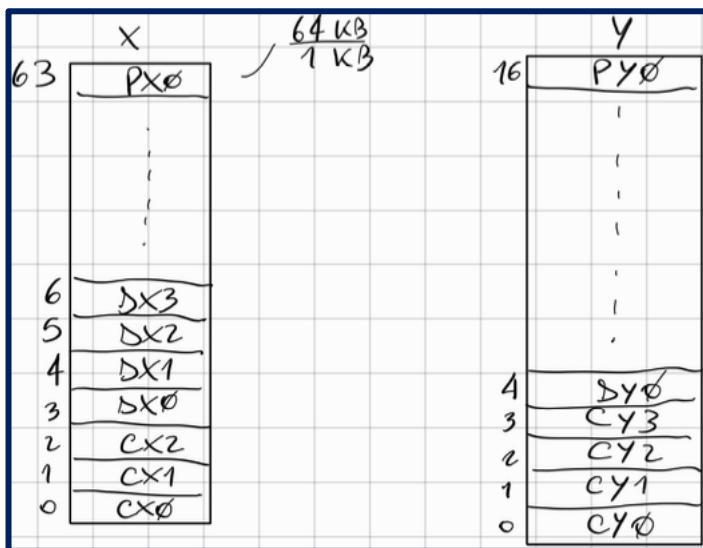
- $NPF = \log_2 \left(\frac{Mem_{phy}}{\text{pagina}} \right)$ [bit]
- $NPV = \log_2 \left(\frac{Mem_{virt}}{\text{pagina}} \right)$ [bit]
- $OFFSET = \log_2(\text{pagina})$ [bit]

Mentre la seconda parte prevede la compilazione della TLB (tabella) a seconda di quanto specificato nel testo:

viene dato l'ordine di esecuzione di un programma, specificando quali processi compiono certe azioni. Sarà necessario effettuare calcoli in esadecimale per determinare quale pagina virtuale si dovrà caricare/scaricare, di seguito esempi e formule.

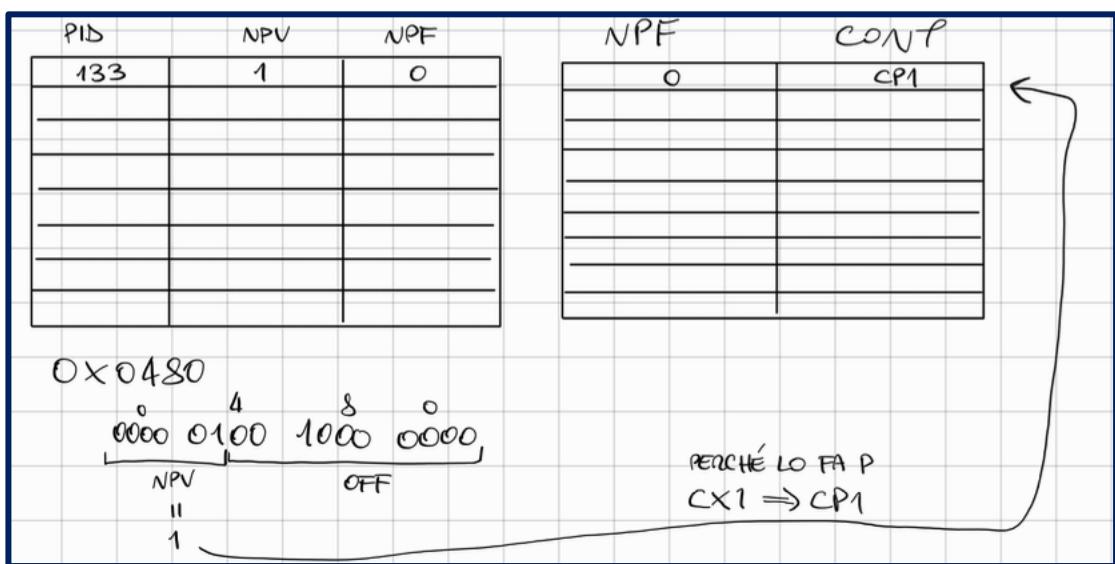


2) PHY MEM = 16 kB	$\rightarrow 14$ bit	$\Rightarrow NPV = 16 - 10 = 6$
VIRT MEM = 64 kB	$\rightarrow 16$ bit	$\Rightarrow NPF = 14 - 10 = 4$
PAGE = 1 K	$\rightarrow 10$ bit = OFFSET	
TLB = 8 RIGHE		
TLB SOST = FIFO		
PHY ALLOCATION = DAL BASSO		
R = 4 n° di pagine residenti		
PROGRAMMI		
X: CX = 3K, DX = 4K, PX = 1K 0X0480		C = codice
Y: CY = 4K SY = 1K PY = 1K 0X0C10		D = dati
		P = pila



Lo schema dello stack è necessario per risolvere l'esercizio, altrimenti non si riuscirà a determinare quale pagina si sta caricando in memoria.

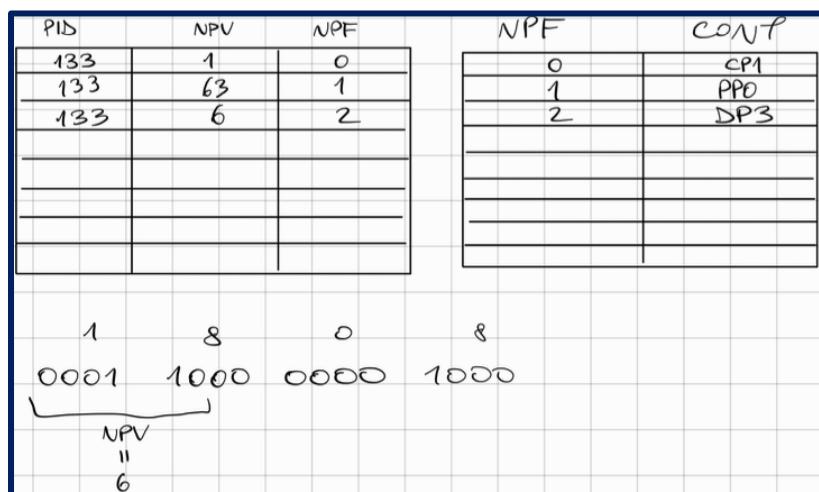
A inizio processo il sistema carica una pagina di text ed una di pila. Viene creato processo P con PID=133 che esegue X.



Quindi il main alloca lo stack

PID	NPV	NPF	NPF	CP1
133	1	0	0	CP1
133	63	1	1	PPO

Ora P accede ad una variabile globale all'indirizzo 0x1808



Il processo P crea un processo Q con PID 266 utilizzando meccanismo di lazy loading

PID	NPV	NPF	NPF	CONT
133	1	0	0	CP1 / CQ1
133	63	1	1	PP0
133	6	2	2	DP3 / DQ3
266	1	0	3	PQ0
266	6	2		
266	63	3		

CONDIVIDO 1^a PAG
DI TEXT

CONDIVIDO 1^a PAG
DI DATI

Il processo Q scrive all'indirizzo 0x1820 $\Rightarrow NPV = 6$

PID	NPV	NPF	NPF	CONT
133	1	0	0	CP1 / CQ1
133	63	1	1	PP0
133	6	2	2	DP3 / DQ3
266	1	0	3	PQ0
266	6	2		
266	63	3		

PID	NPV	NPF	NPF	CONT
133	1	0	0	CP1 / CQ1
133	63	1	1	PP0
133	6	2	2	DP3
266	1	0	3	PQ0
266	6	4	4	DQ3
266	63	3		

Nota: avevo due pagine che condividevano una pagina fisica, poiché su una ho scritto allora le divido.

Il processo Q esegue la sostituzione del codice con Y

P	Q	PID	NPV	NPF	NPF	CONT
X	Y	133	1	0	0	CP1
PX0	PY0	133	63	1	1	PP0
.	.	133	6	2	2	DP3
:	:	266	3	3	3	DQ3
DX3	SY0					
DX2	CY3					
DX1	CY2					
BX0	CY1					
CX2	CY0					
CX1						
CX0						

$Y \rightarrow 0x0C10 \Rightarrow NPV = 3$

Allocò lo stack

PID	NPU	NPF		NPF	CONT
133	1	0		0	CP1
133	63	1		1	PP0
133	6	2		2	DP3
266	3	3		3	CQ3
266	63	4		4	PQ0

Il processo P termina l'esecuzione

PID	NPU	NPF		NPF	CONT
				1	1
				1	1
				1	1
				3	CQ3
266	3	3		4	PQ0
266	63	4			

PID	NPU	NPF		NPF	CONT
				1	1
				1	1
				1	1
				3	CQ3
				4	PQ0

Q sta usando 640 byte di stack

Q contiene:

$f() : \{ \text{int} \times [800] \}$

STACK Q \rightarrow 640

ALLOCO 800 INTERI SULLO STACK $\rightarrow 640 + 800 \cdot 4 = 3840$

\Rightarrow MI SERVONO 4 PAGINE ($3840 / 1024 = 4$)

Q

Y

Py0
Py1
Py2
Py3

↓

Le alloco uno alla volta

1) 266 62 0
2) 266 61 1
3) 266 60 2
4) 266 3 3
5) 266 63 4

↓

R = 4 !

↓

Cancello secondo FIFO

266 62 0
266 61 1
266 60 2
266 3 3
266 63 4

↓

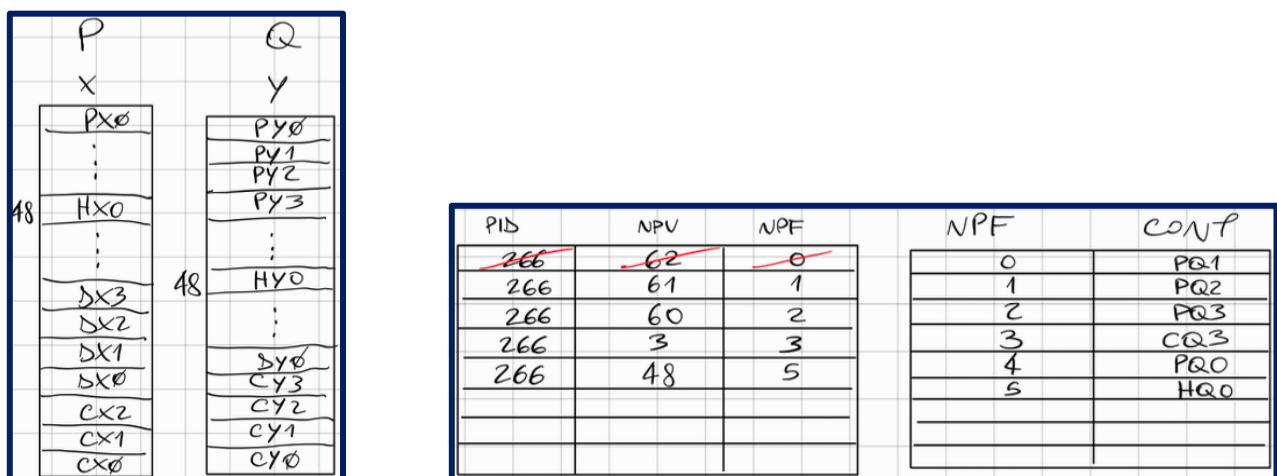
0 1 2 3 4

CONT

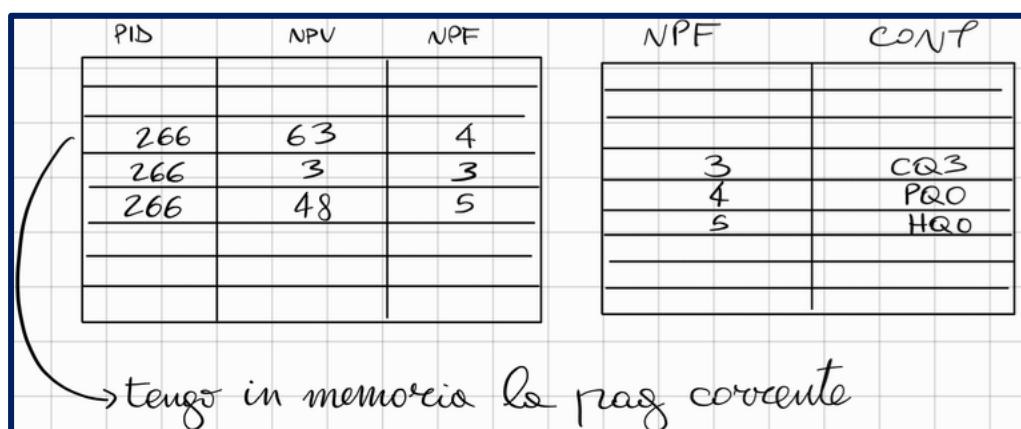
0 PQ1
1 PQ2
2 PQ3
3 CQ3
4 PQ0

Dovrò tenere pagine di text e stack

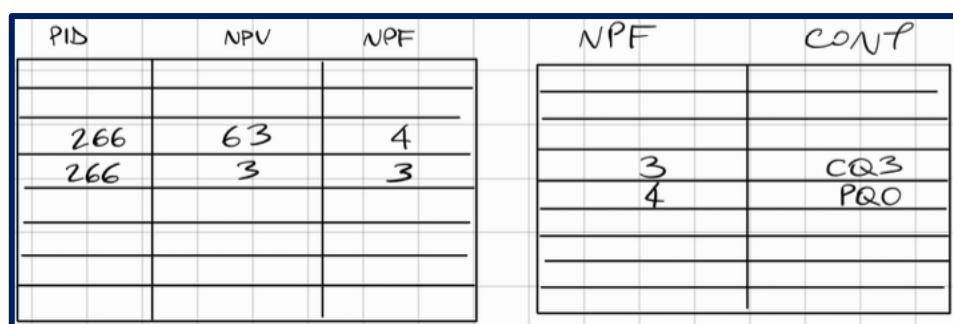
Viene effettuata una malloc($64 * \text{sizeof(double)}$)=512 B, quindi una pagina



Con la return si deve deallocare lo stack



Si chiama il free(...) per la malloc



Sistema Operativo – Filesystem

Il *Filesystem* è un modulo del sistema operativo che fornisce astrazione ai meccanismi di memorizzazione di massa, si basa su *file*, *directory* (*cartelle*) e *volume*.

File: insieme di informazioni omogenee, indipendente dal supporto di memoria su cui è salvato; ad esso sono associati attributi e operazioni eseguibili, come le seguenti.

- **Attributi:** nome, tipo, locazione, dimensione, protezione, proprietario, ora e data (ultima modifica/creazione)
- **Operazioni:** creazione, scrittura, lettura, spostamento, cancellazione
 - ↳ *allocazione*
↳ *creazione descrittore*
↳ *aggiungi descrittore*
 - ↳ *su file già creato*
 - ↳ *preleva dati da file*
 - ↳ *sposta puntatori sia di lettura che scrittura*
 - ↳ *elimina file*
↳ *rimuove descrittore e dealloca spazio*

Per rendere più efficiente l'accesso ai file per le operazioni appena elencate, il filesystem sfrutta una tabella dei file attualmente in uso.

Servizi di base: servizi offerti dall'OS, in particolare sono *OPEN* e *CLOSE*.

- *OPEN* → dal nome individua la posizione del file sul disco, copia il descrittore nella tabella del FS
- *CLOSE* → da ID localizza descrittore del file in tabella e lo elimina

Protezione: può essere realizzata sia a livello fisico utilizzando i *backup* (salvataggio periodico dei dati su supporto esterno/secondario) ed il *mirroring* (supporti di memoria ridondanti) o a livello logico impedendo gli accessi impropri mediante definizione e implementazione di opportuna politiche d'accesso.

Accesso controllato: basato su *identità/gruppo di lavoro* dell'utente e sulle proprietà del file; dipende dai permessi associati ai file, che determinano quali operazioni si possono eseguire e quali no.

Lista d'accesso: associata ad ogni file, indica quali operazioni sono consentite e a quali utenti o gruppi di utenti.³⁶

Owner	Group	Others
rw-	rw-	rw-

↓
Execute
↓
Write
↓
Read

Protezione in UNIX: più semplice del precedente, dove gli utenti sono identificati da nome e gruppo; dal punto di vista delle proprietà, gli utenti sono raggruppati in *owner*, *group* e *others*. La lista degli accessi risultante è quindi formata da tre gruppi di bit.

Osservazione: il filesystem ha sia una struttura fisica che una logica, di seguito approfondite.

³⁶ Gli svantaggi sono dovuti alle dimensioni del sistema, che potrebbero rendere inefficiente questo meccanismo.

Struttura fisica

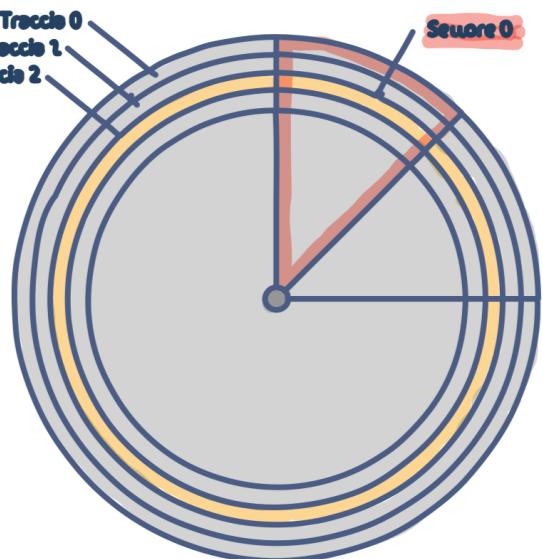
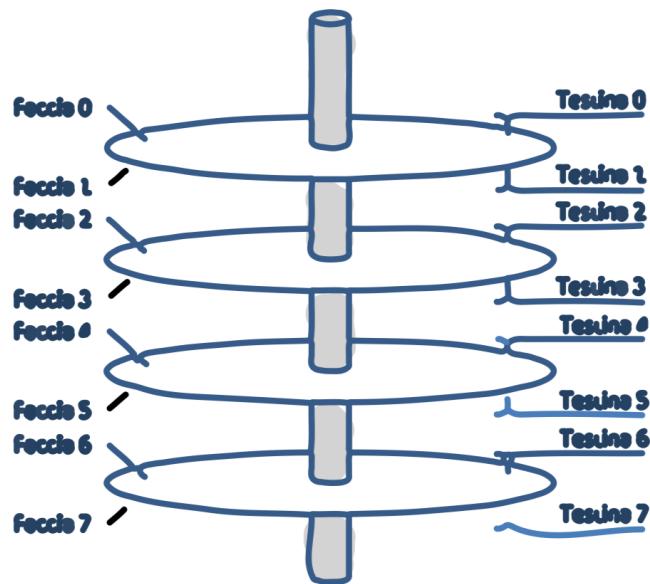
I dati sono organizzati in una sequenza di blocchi a lunghezza fissa (da $32B \rightarrow 4096B$), dipendente dal dispositivo.

Frammentazione interna: fenomeno provocato dalla dimensione fissa dei blocchi, potrebbe portare ad avere blocchi più piccoli della dimensione richiesta (per esempio, da un programma).

Dispositivo di archiviazione: un esempio è l'HDD, essenzialmente un disco magnetico strutturato nel seguente modo.



In realtà non si usa un singolo disco, ma un insieme di questi dischi su più livelli; questi sono posti in rotazione a regimi elevati, sfruttando una testina magnetica per leggere e scrivere i bit.

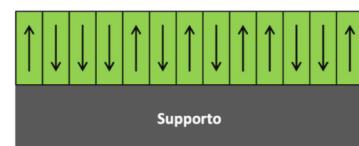


Orientamento del campo magnetico ↗



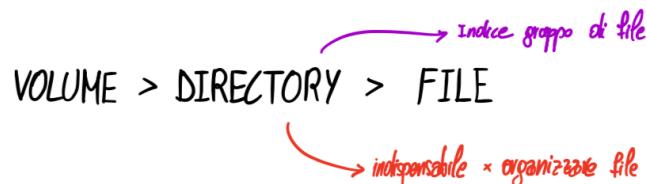
Orientamento Longitudinale

Orientamento Verticale



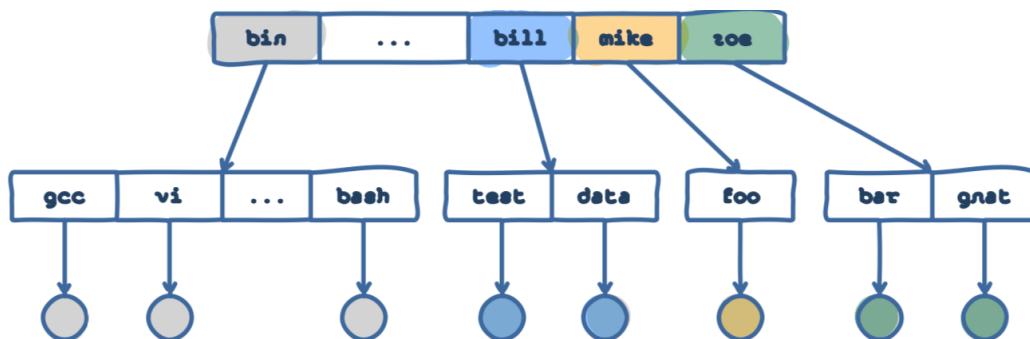
Struttura logica

Organizzazione: il supporto scelto adotta, a livello logico, una struttura gerarchica composta essenzialmente da 3 elementi.



Directory: su di esse si possono compiere operazioni di *ricerca, creazione, rimozione, elencare file, modifica proprietà del file*. Si possono distinguere diverse tipologie, in base alla “topologia” o gerarchia adottata, come illustrato di seguito.

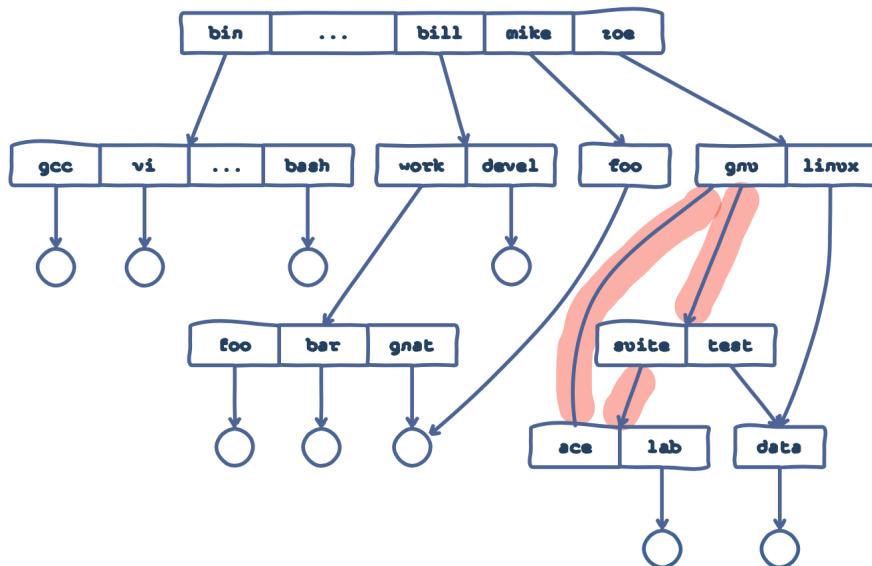
- *Singolo livello*, prevede che tutti i file siano contenuti in una singola directory, peggiora all'aumentare del numero di file
- *Due livelli*, dove vi è una directory principale che contiene le directories degli utenti, ciascuno dei quali gestisce i file nella sua directory; invece, la *root directory* è gestita solamente dal sistema, per le applicazioni vi è un'apposita cartelle



- *Ad albero*, come il precedente, si differenzia per avere l'accesso tramite *pathname* del file; introduce il concetto di *cwd*³⁷, in cui i file si possono cercare indicando solamente il nome (e non tutto il percorso).
- *Grafo aciclico*, per condividere 1+ file tra diversi utenti
Ad esempio, UNIX usa i riferimenti, distinguendoli:
 - *Soft link*, quando si rimuove un file, i riferimenti ad esso puntano a *null*
 - *Hard link*, il file viene rimosso se e solo se tutti i riferimenti ad esso vengono rimossi.
- *Grafo generale*, particolare struttura che può avere riferimenti circolari, è molto flessibile ma complica ulteriormente il sistema operativo.

✓ Per riferimento (venerato desolito)
✗ Per duplicazione (copia locale in altri nodi)

³⁷ Acronimo di Current Working Directory.



Implementazione

Il ruolo del sistema operativo consiste nel fornire la visione logica del filesystem, che è un suo modulo, strutturato a livelli; al gradino più basso vi sono i dispositivi hardware, come ad esempio i supporti di memoria di massa precedentemente visti.

Applicazione	Richiedono funzioni al file system logico.
File System Logico	Sulla base del nome di un file e dell'organizzazione delle directory, genera richieste al modulo per l'organizzazione dei file. Legge i descrittori dei file restituendo la posizione.
Modulo di Organizzazione dei file	Conosce l'organizzazione logica e fisica. Traduce le richieste logiche in richieste fisiche verso il file system di base. Genera il numero assoluto di un blocco dato il suo numero relativo all'inizio del file.
File System Fisico	Invia comandi generali alla parte di controllo dell'I/O. Dato il numero di blocco assoluto genera informazioni specifiche quali faccia, settore, traccia, blocco fisico.
Controllo dell'I/O	Generano i segnali di controllo a partire dai comandi ricevuti. Tali programmi prendono il nome di driver.
Dispositivi	Eseguono i comandi richiesti attraverso i driver

38

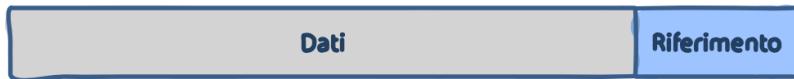
Allocazione contigua: prevede la memorizzazione dei file in blocchi adiacenti, i tempi di accesso sono ridotti ma si hanno problemi nell'allocazione di spazio per i nuovi file → nel tempo si formano zone inutilizzate di piccole dimensioni³⁹.

- *Soluzioni*, per ridurre il problema si può usare la compattazione dei dischi (raggruppare queste zone e unirle) oppure si memorizza il file in zone differenti utilizzando una parte di zona aggiuntiva detta *extent*.

³⁸ Schema in cui sono rappresentati i livelli del filesystem, raggruppati per funzionalità.

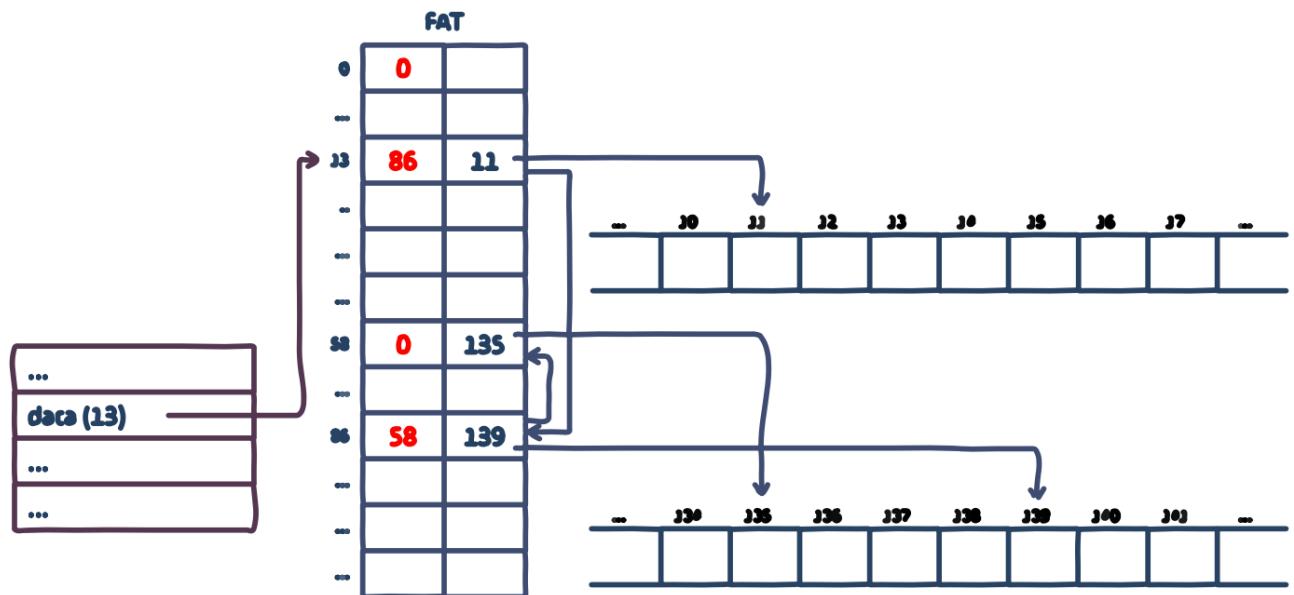
³⁹ È il fenomeno di *Frammentazione Esterna*.

Allocazione concatenata: estensione dell'uso degli extent, dove ogni blocco contiene il riferimento al successivo; risolve la *frammentazione esterna* ma riduce -leggermente- lo spazio disponibile per i dati.



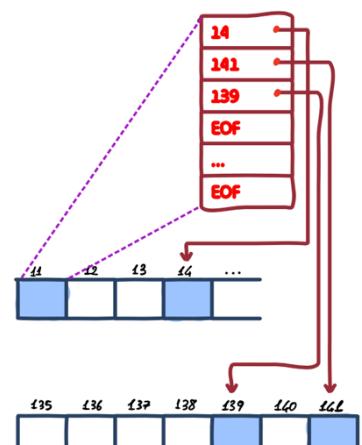
Cluster: raggruppamento di blocchi, risolve le problematiche dell'allocazione concatenata; prevede l'accesso a tutto il gruppo di blocchi; quindi, non vi è bisogno di cercare singoli blocchi. Il problema in questo caso è il potenziale aumento della frammentazione interna.

FAT: ulteriore soluzione che prevede l'utilizzo della *File Allocation Table* (FAT) contiene tanti elementi quanti blocchi del disco; ogni elemento della tabella contiene l'indice dell'elemento FAT che contiene il blocco successivo. Di seguito uno schema che ne illustra il funzionamento.



Allocazione indicizzata: basata sul raggruppamento di tutti i riferimenti, contenuti in un blocco *indice* risolvendo la scarsa efficienza delle precedenti soluzioni. Elimina la frammentazione esterna a discapito di una riduzione della memoria (almeno un blocco per indice, solitamente coincide con un blocco fisico).

- *Schema concatenato*, variante dove l'ultimo elemento del blocco indice contiene il riferimento al secondo blocco indice
- *Schema multilivello*, prevede un blocco indice che contiene riferimenti ai blocchi indice di secondo livello, i quali a loro volta potranno contenere riferimenti ad altri blocchi indici oppure ai blocchi di dati (ma non entrambi)



- Schema combinato, solo la parte finale del blocco indice contiene riferimenti ai blocchi indice di secondo livello

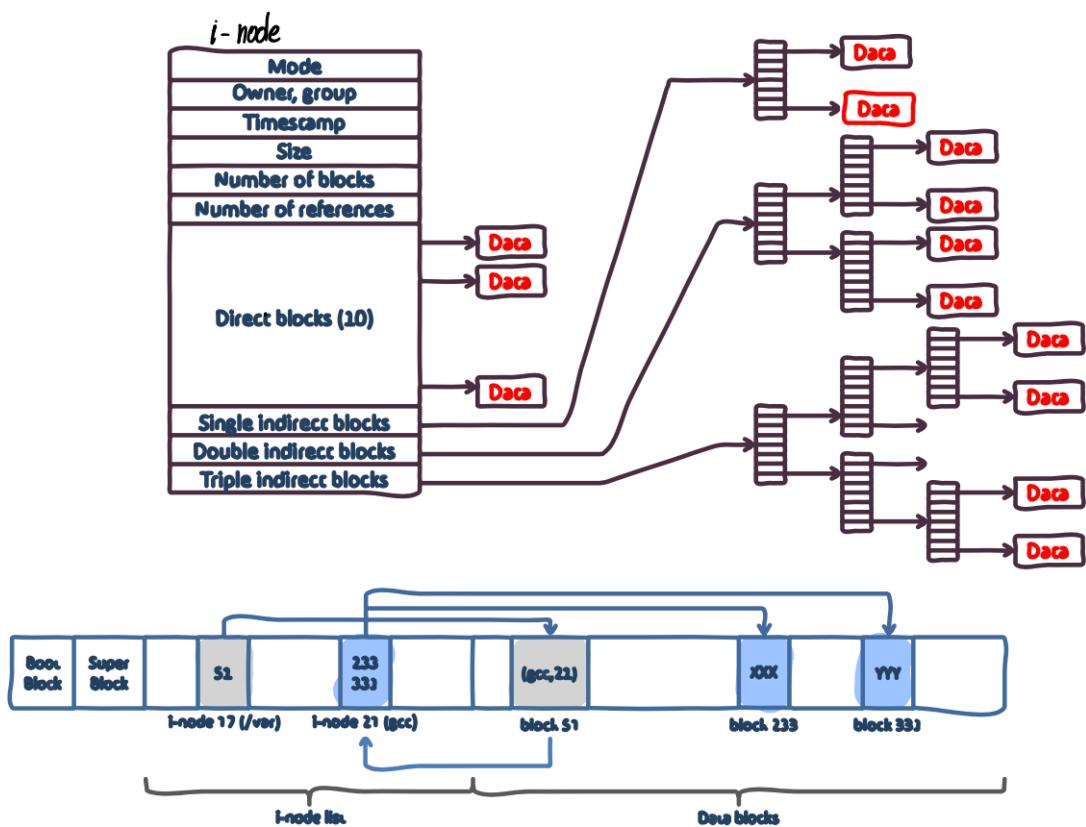
Gestione dello spazio libero: per individuare un blocco libero si può utilizzare un vettore di bit (una posizione per blocco) dove ‘0’ indica occupato ed ‘1’ libero. Un’altra soluzione potrebbe essere rappresentata da una lista concatenata con FAT o *Raggruppamenti*.

Realizzazione directory: a partire dalla lista contenente nomi e descrittori dei file, anche se poco efficiente; una soluzione migliore consiste nell’utilizzo delle tabelle di *hash*, costituite da un vettore di descrittori più la funzione di *hash*, che restituisce l’indice del descrittore cercato.

Filesystem Linux

Ha una struttura organizzata in volumi, costituito ognuno da 4 zone principali, in particolare:

1. *Blocco di boot*, contiene il codice di inizializzazione dell’SO
2. *Superblock*, contiene le informazioni globali del filesystem, generalmente informazioni relative alla sua gestione
3. *Lista i-node*, contiene *i-node* (descrittori) necessari ad indicizzare i file
4. *Blocchi di dati*, contiene i dati dei file e le directory

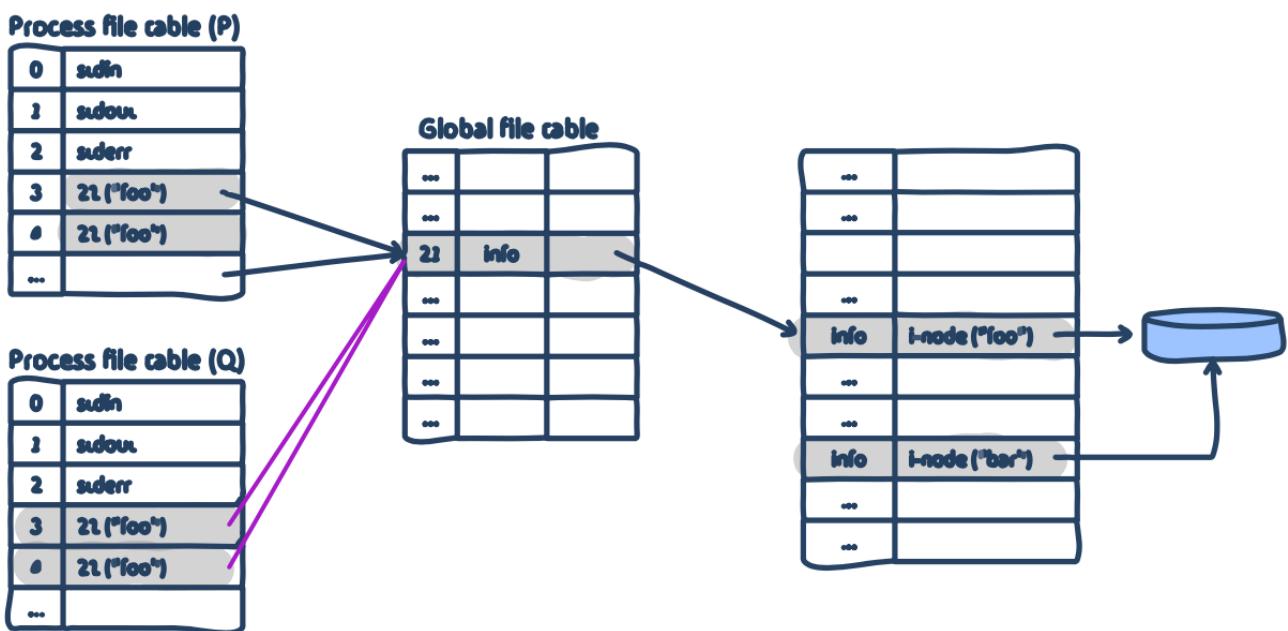


Formattazione: è l'operazione che rende utilizzabile il disco dandogli una struttura logica (*mkfs*, comando), segue tappe ben precise.

- Vengono scritte le dimensioni del volume e delle liste nel superblock
- Viene allocata la *i-list* di dimensione fissa
- Gli *i-node* sono disposti in sequenza
- Viene costruita la lista dei blocchi liberi

Gestione concorrente: permette al sistema operativo di gestire filesystem per conto di diversi processi concorrenti, ma saranno necessarie informazioni aggiuntive, contenute nelle tabelle descritte di seguito.

- *Tabella dei file aperti per processo*⁴⁰, associata ad ogni processo attivo nel sistema operativo, è parte della sezione *system data* dell'immagine del processo. Ogni entry contiene l'elemento associato ad un file aperto, l'indice della tabella dell'identificatore del file, l'elemento che contiene l'indice nella tabella globale.
- *Tabella globale dei file aperti*, contiene i riferimenti a tutti i file attualmente aperti (da processi attivi); ogni riga contiene l'elemento associato al file, il puntatore al suo *i-node*, indice della posizione corrente (nel file), contatore dei riferimenti, modalità di apertura.
- *Tabella i-node*, raccoglie gli *i-node* e le informazioni sul suo stato.



⁴⁰ Per convenzione sono sempre aperti lo standard input, standard output e lo standard error.

Primitive di gestione: del filesystem, si possono distinguere in quelle *di base* (accesso a basso livello) oppure in *avanzate* (accesso con stream); mentre la gestione di un file richiede l'esecuzione di alcune operazioni.

- *Apertura*, creazione elementi nelle tabelle, caricamento degli *i-node*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open( const char *pathname, int flags, mode_t mode );
```

- Accesso, creazione/lettura e scrittura sequenziale/riposizionamento per accesso casuale/cancellazione

```
#include <unistd.h>
ssize_t read( int fd, void *buf, size_t count );
```

```
#include <unistd.h>
ssize_t write( int fd, void *buf, size_t count );
```

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

- *Chiusura*, prevede il rilascio delle risorse e l'aggiornamento delle tabelle

```
#include <unistd.h>
int close( int fd );
```

Sistema Operativo – Driver

Il *driver* è un modulo del sistema operativo esterno al kernel, dedicato alla gestione di una specifica periferica: il suo compito è fornirne una virtualizzazione. Interagisce con le applicazioni degli utenti, il filesystem ed il kernel.

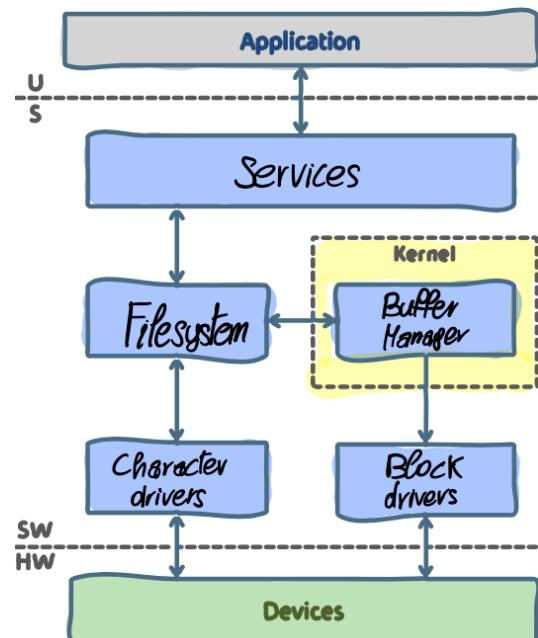
Periferiche: in ambienti UNIX sono distinte in *periferiche a carattere* che adottano l'accesso sequenziale, ed a *blocchi* che supportano l'accesso casuale (necessaria l'indicizzazione).

La distinzione tra periferiche porta ad avere la stessa classificazione anche nei *driver*, infatti si avranno:

- *Driver a carattere*, interagiscono in modo diretto col filesystem
- *Driver a blocchi*, interagiscono col filesystem tramite buffer di sistema

File speciali: associati a ogni periferica, contenuti nella directory “/dev”, mentre l’*i-node* associato indica tipo e specifica della periferica/driver.

Accesso a dispositivo: l’accesso al codice è effettuato proprio attraverso il file e fornendo il buffer per la “comunicazione”; di seguito uno spezzone di codice d’esempio.



```
...
fd = open( "/dev/lp0", O_WRONLY );
write( fd, buffer, buffer_length );
close( fd );
...
```

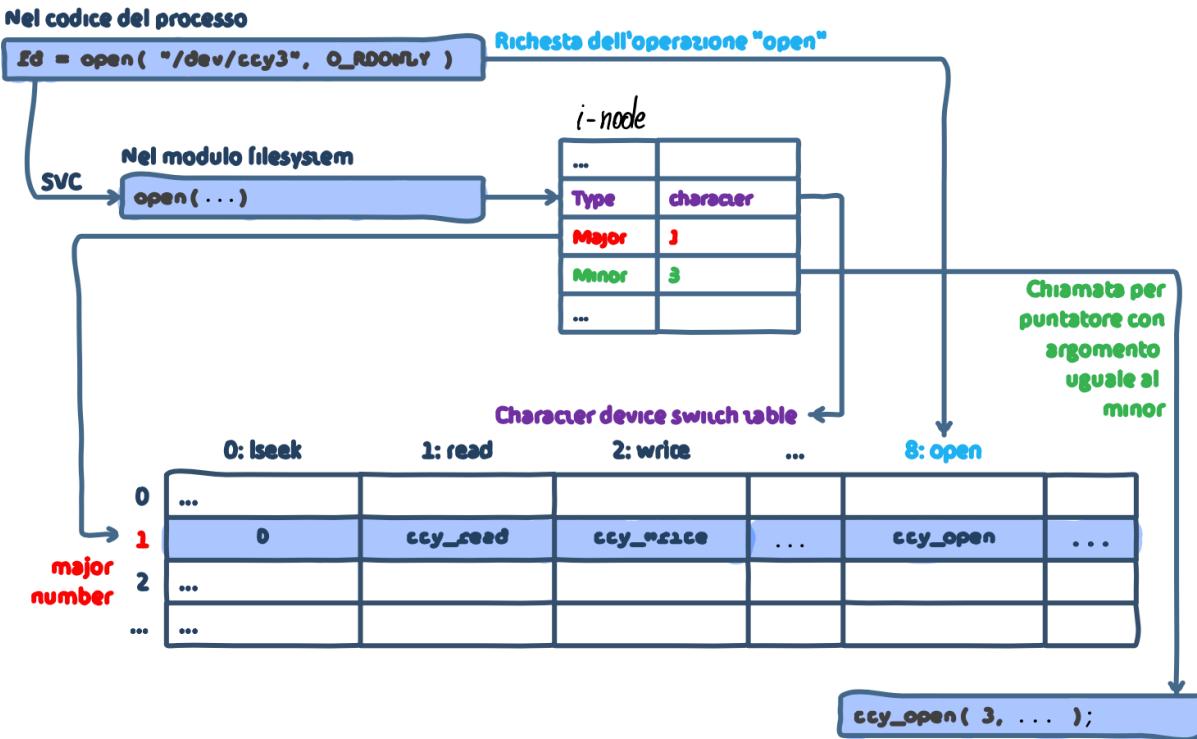
File operations: struttura dati che realizza il meccanismo d’accesso omogeneo per gestire ogni periferica con un’interfaccia comune; è essenzialmente un insieme di puntatori a funzione. Il meccanismo d’accesso segue tappe ben definite

1. inizializzazione
2. Si invoca funzione speciale per ogni driver
3. OS copia i puntatori alle funzioni in tabella
4. Il *major number* identifica la riga della tabella (device)
5. In base all’operazione richiesta si sceglie la colonna
6. Tramite il puntatore si accede alla funzione del driver (riga x colonna scelte)

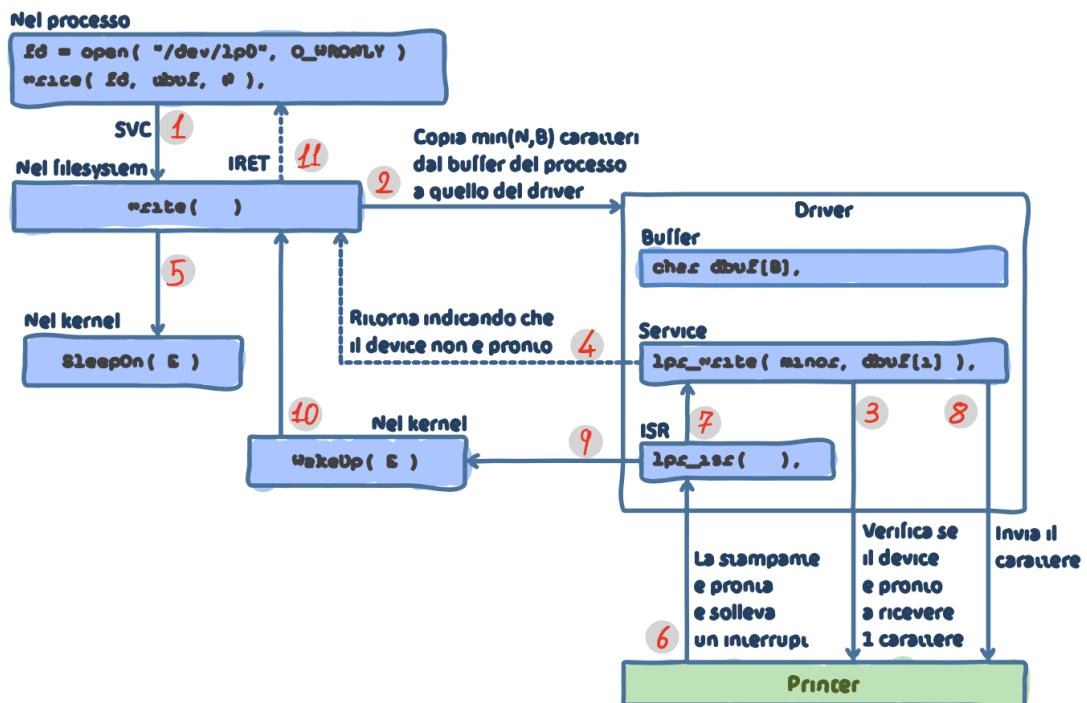
```
struct file_operations {
    int (*lseek)( );
    int (*read)( );
    int (*write)( );
    ...
    int (*ioctl)( );
    int (*open)( );
    void (*release)( );
    ...
};
```

Osservazione: il *major number* indica il tipo di periferica/driver, mentre il *minor number* indica la specifica periferica tra tutte quelle dello stesso tipo disponibile nel sistema.

In maniera più schematica sono illustrate di seguito le operazioni per l'accesso.



Accesso a carattere: nel caso in cui la periferica a carattere sia gestita tramite interrupt si avrà una situazione come la seguente.



Altrimenti, sfruttando il *direct memory access* (DMA) si dovranno eseguire le operazioni specificate nello schema sottostante. È una modalità d'accesso particolarmente usata nelle periferiche a blocchi.

