

05/03/2021

Introduzione ai sistemi operativi

Cos'è un sistema operativo? → programma che agisce da intermediario tra l'utente e l'hardware dell'elaboratore!

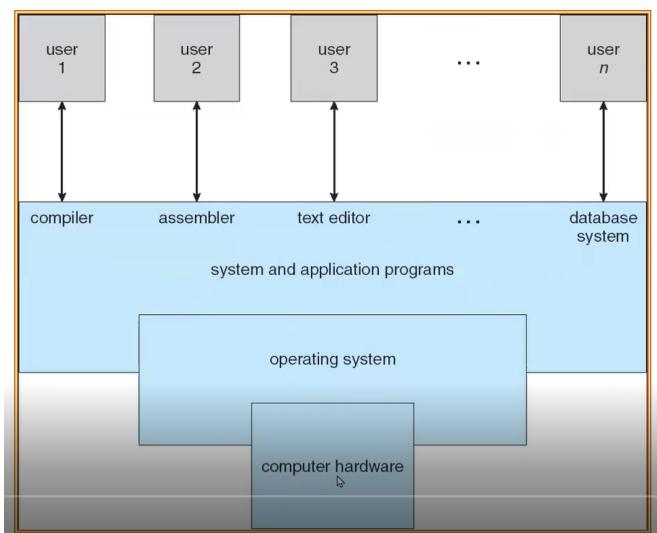
→ inoltre lo deve fare nel modo migliore possibile!

Gli obiettivi di un sistema operativo saranno quindi:

- Eseguire i programmi degli utenti e facilitare la risoluzione dei problemi degli utenti
- Rendere conveniente l'uso dei calcolatori e usare l'hardware del calcolatore in modo + efficiente possibile in modo da garantire l'esecuzione sicura di tutto ciò che viene passato!

+Un calcolatore qualsiasi dunque può essere diviso in 4 componenti:

- Hardware → che fornisce le componenti di base come la cpu, la memoria e i dispositivi di input/output.
- Sistema operativo → che coordina l'uso dell'hardware tra le varie applicazioni e i vari utenti
- Programmi applicativi → sopra il sistema operativo, usano le risorse per risolvere i problemi computazionali degli utenti → ad esempio i compilatori, web browser o anche i video games...
- Utenti → possono essere persone oppure anche altre macchine o calcolatori!



→ programmi che possono essere o programmi di sistema o programmi applicativi (i primi sono quelli dati dal sistema operativo, gli altri sono acquisiti anche successivamente!)

→ gli utenti usano l'hardware tramite delle applicazioni → queste accedono all'hardware tramite il sistema operativo

+Alcune definizioni...

→ Il Sistema Operativo è un allocatore di risorse:

- Gestisce tutte le risorse del calcolatore

- Decide tra richieste in conflitto per un uso più efficiente possibile e anche giusto delle risorse disponibili → come CPU, memoria o dispositivi di I/O

→ Il Sistema Operativo è un programma di controllo

- Controlla l'esecuzione dei programmi per prevenire errori ed un uso improprio del calcolatore → comunque non esiste una definizione universalmente accettata!

→ Può anche essere visto come "Il programma sempre in esecuzione sul calcolatore" → che è il kernel (nucleo) del SO(sistema operativo).

Ogni altra cosa o è un programma di sistema (venduto con il sistema operativo) o è un programma applicativo!

+Una delle fasi più importanti della vita di un calcolatore è l'avvio! → ciò che avviene all'avvio è Il programma di *bootstrap*, il quale è caricato all'accensione o al reboot(una sorta di reset)

→ Il programma è tipicamente memorizzato in ROM o EEPROM, ed è noto come firmware (fa già parte del calcolatore)

→ questo programma si occupa quindi di inizializzare e controllare tutti gli aspetti del sistema → oggi parte meno evidente! (lasciata al *bios*?)

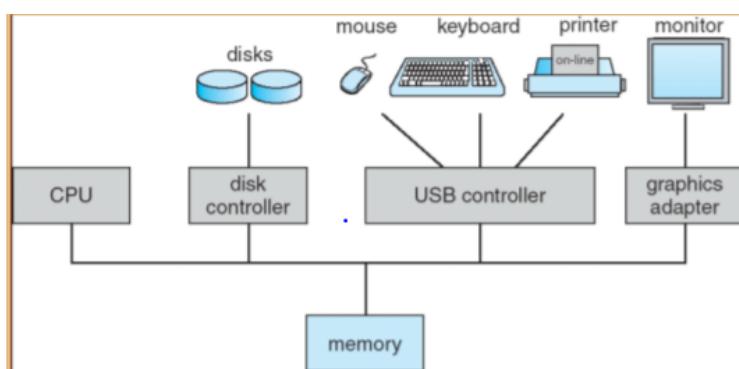
→ successivamente viene caricato il kernel del sistema operativo e inizia la sua esecuzione
→ c'è una fase iniziale di caricamento dei dati e successivamente di tutto il resto!

+ In caso di fallimento nell'avvio il sistema non è utilizzabile! → può fallire ad esempio per un problema hardware sulla macchina oppure può essere un fallimento perché il caricamento del kernel ha avuto un problema precedente → ad esempio la rottura del disco ... e si cerca in qualche modo di recuperare dal problema

+ esempio → il *bootloop* dei telefoni cellulari → il telefono si resetta e riparte! → perchè appunto trova qualcosa che non va bene all'interno del nostro telefono e dunque ogni volta si resetta e cerca di ripartire normalmente → cosa che può portare anche a loop in cui non si esce dunque il telefono è in crash!

+Com'è organizzato un calcolatore?

Nel calcolatore abbiamo una o più CPU, poi altri dispositivi connessi attraverso un bus che fornisce l'accesso ad una memoria condivisa



- l'Esecuzione concorrente delle CPU (infatti spesso non è una sola) e dispositivi che competono per l'uso della memoria (memory cycles) → possono infatti esserci più programmi contemporanei che vorranno accedere ai vari dispositivi e alla memoria per eseguire le varie istruzioni! → ognuna delle esecuzioni compete per l'accesso alla memoria per eseguire un'istruzione
- per cui in generale la CPU non sta sempre ad aspettare la risposta dei dispositivi di input/output, anzi spesso fa altre cose!

+Checkpoint → cos'era un calcolatore? tante cose...

→ Dispositivi I/O e la CPU possono operare in modo concorrente.

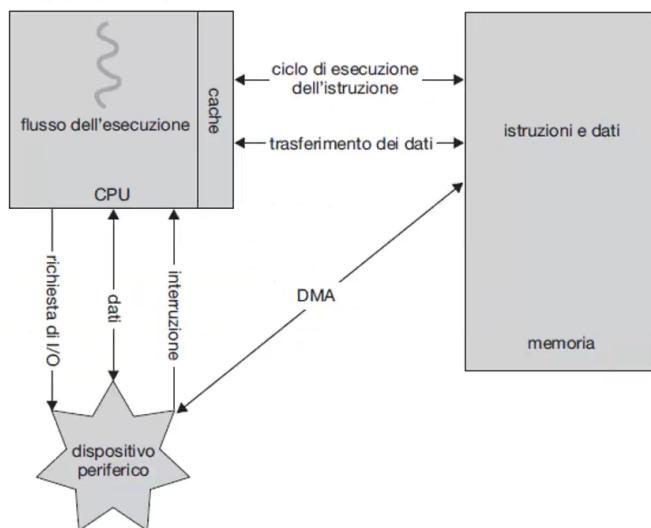
→ Ogni controllore di dispositivo si occupa di un tipo di dispositivo (es. controller del disco).

→ Ogni controllore di dispositivo ha un suo buffer locale. (dove tenere i dati)

→ La CPU sposta dati da/a memoria a/da buffer locali

→ I/O è dal dispositivo al buffer locale del controllore.

→ Il controllore di dispositivi informa la CPU che ha finito la sua operazione causando una interruzione (interrupt).



→ cache all'interno della CPU che permette l'interfaccia alla memoria e dunque il trasferimento dati!

→ oppure c'è la possibilità di dare accesso diretto alla memoria per la periferica tramite il DMA! → bypassando l'accesso alla CPU!

+Funzioni comuni delle interruzioni: Quando e come si usano le interruzioni?

L'interruzione trasferisce il controllo alla routine di gestione della interruzione attraverso il **vettore delle interruzioni** che contiene gli indirizzi di tutte le routine di servizio.

→ L'interruzione deve salvare l'indirizzo dell'istruzione interrotta.

→ Le interruzioni vengono disabilitate quando una interruzione viene gestita per impedire la perdita di interruzioni!

+ Una trap è una interruzione generata dal software causata o da un errore o da una richiesta dell'utente.

Si dice quindi che i sistemi operativi sono guidati dalle interruzioni (interrupt driven) → tramite queste infatti riescono a lavorare in modo più efficiente! → interruzioni usate anche nei programmi per la richiesta dei servizi

+Come agisce quindi il sistema operativo durante la gestione dell'interruzione?

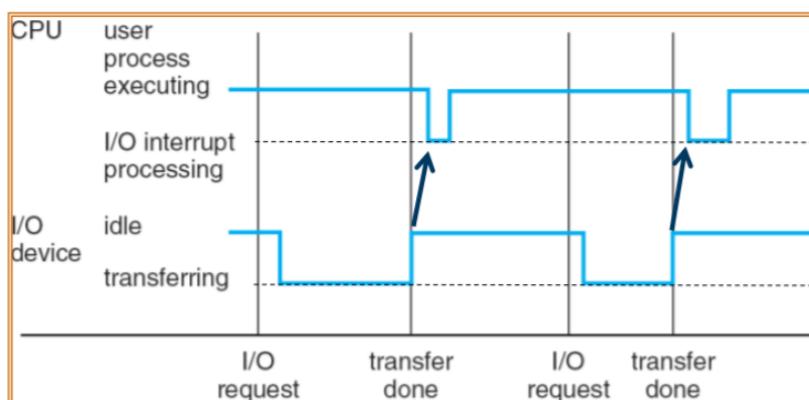
→ questo preserva lo stato della CPU memorizzando il contenuto dei registri e del program counter

→ si determina quindi che l'interruzione è avvenuta attraverso il:

- polling → guardando qual'è il dispositivo (fra tutti) che ha generato l'interruzione (fatta su dispositivi embedded o cose più semplici!) → perdita di tempo!!
- vectored interrupt system → tramite interruzioni vettorizzate → il dispositivo ci dice già chi è che ha generato l'interruzione → e dunque si penserà a come gestirla!

→ Abbiamo quindi segmenti di codice separati determinano che azioni intraprendere per ogni tipo di interrupt!

+timeline interruzione



→ notare che ad ogni trasferimento fatto venga generata un interruzione

→ mentre la cpu è attiva, anche il dispositivo di I/O sta lavorando!

Architettura degli elaboratori

Inizialmente esistevano dei Sistemi monoprocessoressi i quali possiedono una sola CPU per l'esecuzione dei programmi utente, ma possono avere altri processori che svolgono compiti specifici es. controller del disco (ma la loro gestione è interna e dunque non sono gestiti dal sistema operativo!)

Nel caso di Sistemi multiprocessoressi invece abbiamo Più di una unità di elaborazione strettamente connesse, che condividono il bus, il clock e dispositivi di memorizzazione e periferici. → abbiamo più processori fisici e più core all'interno del processore → inteso come scatola fisica

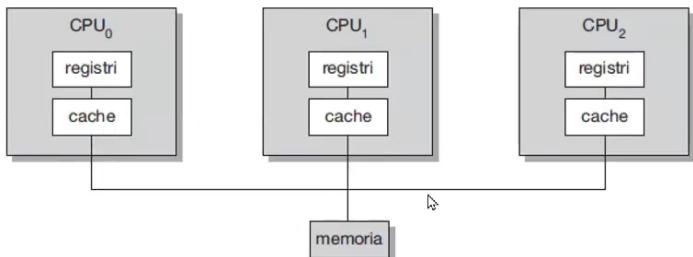
→ Vantaggi:

- Maggiore produttività, in quanto si riescono a fare più cose contemporaneamente
- Economia di scala, condivisione periferiche
- Incremento della affidabilità, un guasto di un processore non blocca il sistema, ma lo rallenta! sistema fault-tolerant → tollerante ai guasti!

+I sistemi multiprocessoressi possono essere quindi di 2 tipi:

- Multielaborazione asimmetrica: in cui ogni unità ha un compito specifico e una unità principale coordina e assegna il lavoro alle altre unità (soluzione poco frequente)

- Multielaborazione simmetrica (SMP), in cui ogni processore è abilitato ad eseguire tutte le funzioni del sistema (soluzione più frequente) → tutti i processori sono in grado di fare tutto! (e non hanno compiti specifici)
- + La tendenza recente è quella di avere più *core* in uno stesso circuito integrato → che porta ad un risparmio energia!



→ notare che ogni cpu ha una propria cache in modo tale da non dover sempre accedere alla memoria → in quanto soprattutto la memoria è unica e dunque l'accesso può essere fatto solo uno alla volta! → per ogni fetch di esecuzione ogni cpu deve aspettare di andare uno alla volta e dunque tempi di latenza molto lunghi!

+Si vedrà in seguito che sono presenti anche cache al di fuori delle CPU → soprattutto per evitare il colloquio con la memoria la quale di solito è l'oggetto più lento con cui comunicare!

Struttura della memoria

Memoria primaria → la sola unità di memorizzazione a cui la CPU puo' accedere direttamente.

Memoria secondaria – estensione della memoria primaria che fornisce una grande capacità di memorizzazione non volatile. (caratteristica infatti della memoria primaria è che è volatile!
→ ovvero se si spegne la macchina il suo contenuto viene perso!)

- + Dischi magnetici (tipo di memoria secondaria)
 - piatti di metallo o vetro coperti da materiale magnetizzabile
- La superficie del disco e' divisa in tracce che sono suddivise in settori. → ogni singolo settore può essere acceduto direttamente attraverso una testina!
- Il controller del disco determina l'interazione tra il dispositivo e il computer → permette quindi l'accesso al disco

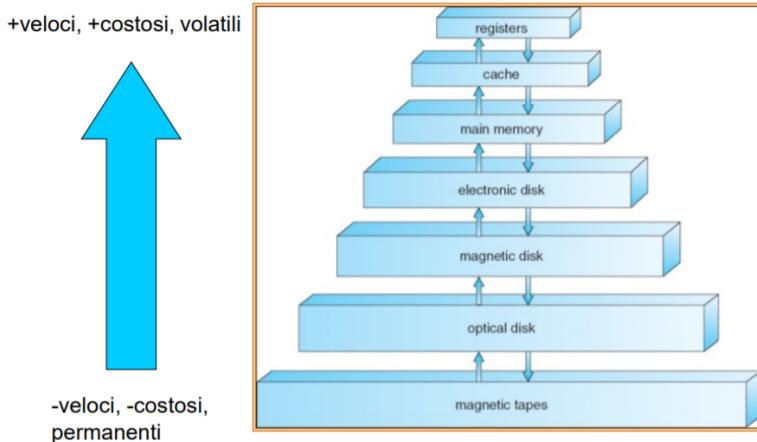
+Sistemi organizzati in una gerarchia, secondo tre aspetti

- Velocità
- Costo
- Volatilità

+L'attività di caching → serve a copiare le informazioni in un sistema di memorizzazione più veloce → da un livello più basso (+ lento) verso una più veloce

→ infatti la mem primaria è vista come una cache per la memoria secondaria

+gerarchia(della memoria)(della memoria)(della memoria):



→ volatili → togliendo a loro l'alimentazione l'informazione viene persa! → da main memory in su tutti volatili! il resto ha la possibilità di mantere la memoria in modo permanente!

→ i magnetic tapes i più capaci (e meno costosi) ma anche i più lenti in quanto bisogna fare accesso alla memoria in modo sequenziale! → per poter accedere ad una dato ci si accede in maniera sequenziale!

Caching

+Il caching non è usato solo a livello hardware, ma anche di sistema operativo o per le applicazioni! → permette infatti di mantere dati in modo da avere un accesso veloce e dunque per rendere più efficiente l'utilizzo del sistema!(anche se ovviamente non è una copia completa dalla memoria principale)(anche se ovviamente non è una copia completa dalla memoria principale)(anche se ovviamente non è una copia completa dalla memoria principale)

→ Informazione in uso viene copiata temporaneamente da dispositivo di memorizzazione lento a uno più veloce.

+come funziona il sistema di caching si dovrebbe sapere da calcolatori... ma si ripete:

→ Memoria veloce (cache) controllata prima per determinare se l'informazione è presente Se lo è, l'informazione è usata direttamente dalla cache (veloce), Altrimenti, dati copiati nella cache (replacing dei dati) e usati da lì !

+La cache di solito è più piccola della memoria dietro la cache → dunque vanno considerati il problema della gestione della cache, ma anche sia la dimensione della cache che della politica di rimpiazzamento! → essendo questa limitata si dovrà infatti decidere ad un certo momento cosa togliere in base alle varie possibilità!

+Movimenti di dati tra livelli della gerarchia possono essere esplicativi (es. caricamento dati da disco) o impliciti (es. cache processore) → di questi ultimi infatti non ce ne rendiamo conto!

→ esecuzione che è trasparente ed è gestita completamente dal processore!

+Performace ai vari livelli...

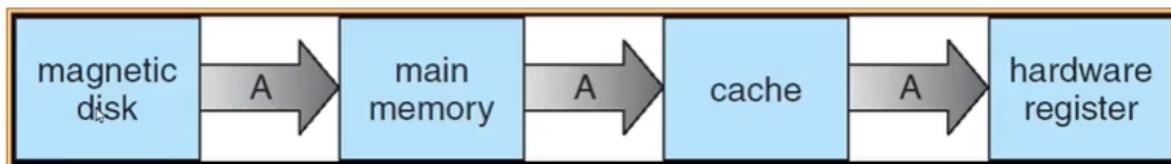
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS CMOS SRAM	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

→ notare il grandissimo passaggio per il tempo di accesso da main memory a disk storage!

—> si passa da nano a milli secondi... —> si passa da nano a milli secondi... —> si passa da nano a milli secondi...

→ dei colli di bottiglia e dunque problemi di performance si hanno quando si accede troppo spesso alla memoria del disco!

+Vediamo quindi come un dato possa migrare fra i vari stratiratirati



→ nel caso di ambienti multitasking → con + attività attive contemporaneamente si deve stare attenti ad usare il valore più recente, indipendentemente di dove è memorizzato nella gerarchia delle memorie → si deve essere sicuri di accedere all'ultima versione del dato!!!
 → Sistemi multiprocessore devono fornire **coerenza** della cache in hardware in modo tale che tutte le CPU abbiano lo stesso valore nella loro cache.

→ nel caso di ambienti distribuiti → + macchine diverse che contengono i dati → e se questi possono essere condivisi fra macchine diverse → collegati tramite una rete allora le cose possono essere ancora in maniera più complessa! → c'è la necessità di tenere aggiornate le varie cache!

→ Possono esistere molte copie del dato in calcolatori diversi → che se per certi casi è una cosa positiva (ad esempio nel caso uno fallisca ce nè un altro disponibile) → ma ho principalmente il problema di tenerle allineate!

Struttura di un sistema operativo Struttura di un sistema operativo Struttura di un sistema operativo Struttura di un sistema operativo

→ la multiprogrammazione è qualcosa di importante per gestire in modo efficiente un calcolatore → in quanto un solo utente non può tenere sempre occupato una CPU e i dispositivi di I/O → infatti vogliamo usare tutti i dispositivi il più possibile! → si vuole avere un ritorno sull'investimento fatto... sarebbe uno spreco lasciare la CPU in wait per troppo tempo(o lo stesso per i dispositivi di I/O)(o lo stesso per i dispositivi di I/O)(o lo stesso per i dispositivi di I/O)

→ La multiprogrammazione gestisce lavori (jobs) (codice e dati) in modo che la CPU abbia sempre qualcosa da fare...

→ non si fa multiprogrammazione solo nei sistemi embedded → nei quali dev'essere eseguita una sola funzione o un solo programma!

+Solo un sottoinsieme dei lavori presenti sul sistema è tenuta in memoria (ogni job infatti ha bisogno di una certa quantità di memoria per essere eseguito!)

+ Un lavoro viene selezionato ed eseguito attraverso il *job scheduling* → ovvero viene deciso quale job deve essere eseguito dal sistema! ! → Quando deve aspettare (per I/O per esempio), il sistema operativo commuta su un altro job. In sostanza il job scheduler deve scegliere (in base alla politica adottata) quale job schedulare a seconda anche dei requisiti richiesti... In sostanza il job scheduler deve scegliere (in base alla politica adottata) quale job schedulare a seconda anche dei requisiti richiesti... In sostanza il job scheduler deve scegliere (in base alla politica adottata) quale job schedulare a seconda anche dei requisiti richiesti...

→ abbiamo la possibilità quindi del *timesharing*(multitasking) → la cpu cambia lavoro così frequentemente che l'utente può interagire con il job mentre questo è in esecuzione → dando la possibilità di avere l'interactive computing! (computazione interattiva ?) —> tutti i processi possono rispondere in un tempo ragionevole —> tutti i processi possono rispondere in un tempo ragionevole —> tutti i processi possono rispondere in un tempo ragionevole → gli utenti possono colloquiare direttamente con il sistema in esecuzione potendo dare direttamente un input e ottenendo di conseguenza un output!

→ questo è però possibile quando il tempo di risposta è inferiore a 1

+Ogni utente ha almeno un programma in esecuzione in memoria → detto *processo* in esecuzione!

→ se invece molti processi sono pronti all'esecuzione allo stesso tempo si ha il così detto *CPU scheduling* → uso della cpu da parte di questi processi → in modo da ottenere l'output in tempi ragionevoli!

→ se ancora invece non c'è abbastanza memoria si ha il così detto *swapping* → si può togliere dalla memoria qualcosa per lasciare spazio ad altri...

→ si tira fuori tutto il processo dalla memoria, ad esempio processi non direttamente usati dall'utente umano e quindi lascio spazio a processi interattivi che necessitano di una risposta più immediata

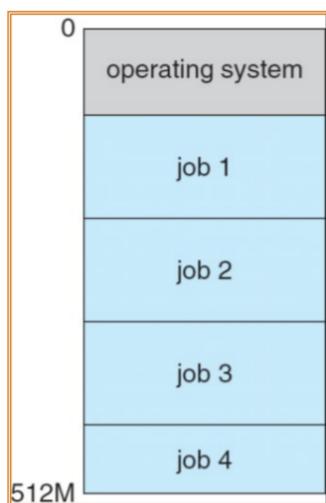
→ infine La memoria virtuale permette l'esecuzione di processi anche non completamente in memoria (ovvero non abbiamo tutta la memoria a disposizione per un processo e dunque gliene carichiamo una parte...) abbiamo tutta la memoria a disposizione per un processo e dunque gliene carichiamo una parte...) abbiamo tutta la memoria a disposizione per un processo e dunque gliene carichiamo una parte...) → fa finta di avere in memoria più spazio disponibile di quello che è realmente → e dunque i processi usano solo la parte di memoria che a loro serve! → ad esempio per i processi di errori → questi verranno appunto invocati solamente se ne abbiamo veramente bisogno → dunque non si riserva loro molto spazio (potrebbe anche essere mai eseguito!)

+Esistono dunque dei metodi ad esempio per caricare il minimo necessario e poi via via che venga richiesto se ne carica dell'altra (ovviamente questo comporta una perdita di tempo! → rallentamento in avvio del programma!)

→ oppure ancora se ad esempio per un processo non più utilizzato e improvvisamente viene richiesto... per questo ci sarà bisogno del tempo per ricaricarlo in quanto se non più utilizzato la memoria riservata al processo è stata sostituita per quella di altri processi! → e dunque via via che si riutilizza il processo questo risponde in tempi ragionevoli!

10/03/2021

→ schema della memoria in un sistema multiprogrammato



Parte della memoria deve essere lasciata al sistema operativo → di solito questo è vicino al vettore delle interruzioni...

Parte della memoria deve essere lasciata al sistema operativo → di solito questo è vicino al vettore delle interruzioni...

Parte della memoria deve essere lasciata al sistema operativo → di solito questo è vicino al vettore delle interruzioni...

Il resto quindi sarà usato dai job in esecuzione! resterà quindi sarà usato dai job in esecuzione

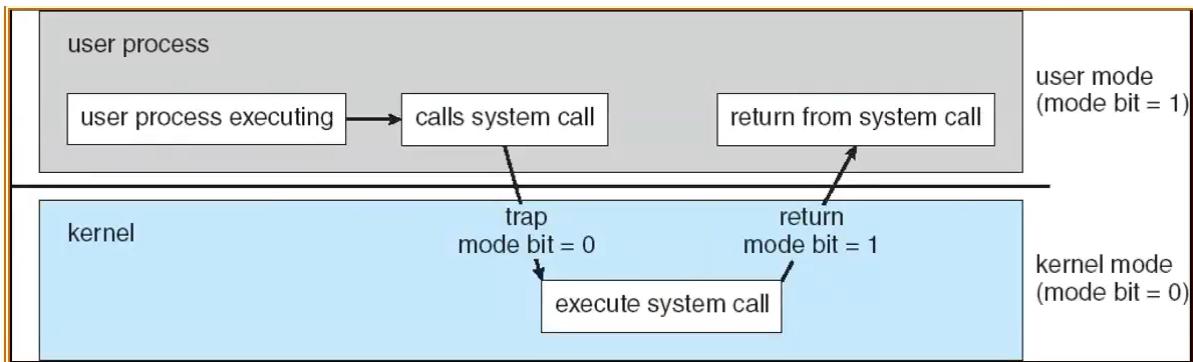
Continuo sistemi operativi

Un sistema operativo è comunque guidato dalle Interruzioni → le quali oltre a quelle di input/output possono essere dovute anche ad un errore software (ad esempio una divisione per zero), oppure una richiesta di un servizio al sistema operativo!

- sistema operativo che deve essere in grado di gestirle le varie possibili interruzioni
- sistema operativo che deve essere in grado di gestirle le varie possibili interruzioni
- sistema operativo che deve essere in grado di gestirle le varie possibili interruzioni
- Altri problemi dei processi includono: i loop infiniti → la cpu diventa ad uso escluso di questo processo dunque il sistema si blocca! → nel caso il sistema sia condiviso con altri utenti questo non è un problema di poco conto!
- oppure ancora per processi che si modificano a vicenda o modificano il sistema operativo
- ad esempio se fanno riferimento a zone di memoria (condivise) (condivise) (condivise) dove sono presenti degli errori! → oppure si fa riferimento ad altre zone di memoria non previste nel programma, dove potrebbero esserci o processi di altri utenti o altri sistemi operativi!
- potremmo quindi ottenere istruzioni senza senso o anche istruzioni valide ma che non eseguono ciò che era stato richiesto!
- in questi casi il sistema si blocca o inizia ad avere un comportamento tempo variante → il sistema diventa inaffidabile!

+ Per evitare questi problemi è dunque presente il *Dual-mode* → User mode e kernel mode

→ se il sistema si trova nel primo → il sistema stà eseguendo del codice dell'utente, mentre nell'altro è in esecuzione il codice del sistema operativo o del kernel
 → questo dunque permette al SO di proteggersi e proteggere altri componenti del sistema
 + Bit di modo, che è fornito dall'hardware, Permette di distinguere quando il sistema sta eseguendo codice utente o codice del kernel
 → i processori di oggi hanno più bit di modo → in modo da codificare più modi, ma almeno uno deve essere presente!
 + Inoltre Alcune istruzioni designate come **privilegiate** sono eseguibili solo in modo kernel
 → ad esempio le istruzione di I/O viste ad inizio del corso sono considerate come istruzioni privilegiate! → le quali dunque non sono eseguibili in modalità utente se si pensa di farlo il processore genererà un'eccezione → la quale sarà gestita dal sistema operativo che non farà altro che fermare il processo che tentava di eseguire quella determinata istruzione!
 + Le chiamate al sistema cambiano il bit di modo a modo kernel e al termine della chiamata ritorna al modo utente! (Solo la modalità kernel fornisce tutte le potenzialità di un calcolatore!) (Solo la modalità kernel fornisce tutte le potenzialità di un calcolatore!) (Solo la modalità kernel fornisce tutte le potenzialità di un calcolatore!)



→ figurina del “processo” → facendo la chiamata al sistema dovrà eseguire un qualcosa legata al sistema operativo... per cui si cambia il mode bit → e si entra nella modalità privilegiata in cui si esegue le operazioni del sistema
 Come fare queste chiamate? Sfruttando delle interruzioni che poi verranno gestite andando quindi a chiamare delle procedure sia per gestire l'interruzione che per eseguire ciò che volevamo...

Come fare queste chiamate? Sfruttando delle interruzioni che poi verranno gestite andando quindi a chiamare delle procedure sia per gestire l'interruzione che per eseguire ciò che volevamo...

Come fare queste chiamate? Sfruttando delle interruzioni che poi verranno gestite andando quindi a chiamare delle procedure sia per gestire l'interruzione che per eseguire ciò che volevamo...

+Questa modalità dunque ci risolve alcuni casi, ma non ci risolve il problema del loop infinito!
 → se l'esecuzione rimane nella modalità utente come fa il sistema operativo ad accorgersi del problema (questo è infatti lì che aspetta magari un'interruzione!)

→ si usano quindi i timer! → dispositivo hardware che permette di svegliare il sistema ad intervalli regolari → genera un interrupt ad intervalli regolari

L'interruzione verrà infatti gestita dal kernel per cui entra in esecuzione e si accorge che è passato un tot tempo da quando il processo è in esecuzione e dunque può decidere che quel processo che è in esecuzione deve essere terminato, perché bloccato! e quindi passare all'esecuzione di un altro processo!

- Il Sistema Operativo decrementa un contatore e Quando il contatore raggiunge lo zero genera un interrupt
- Il contatore viene impostato dal sistema operativo prima di *schedulare* un processo, per riottenere il controllo e terminare il processo se occupa più tempo del previsto. → dunque il timer assume un ruolo importante nello scheduling della CPU!

Gestione dei processi/memoria Gestione dei processi/memoria Gestione dei processi/memoria

Un processo è un programma in esecuzione —> entità attiva!e —> entità attiva!e —> entità attiva!

→ Un Programma è una entità passiva, mentre un processo è una entità attiva → ha infatti bisogno di risorse per eseguire quello che lui richiede!

→ Ha bisogno di CPU, memoria, I/O, files (deve infatti leggere, oppure mettere dei dati da qualche parte), oppure dei Dati iniziali e a quel punto inizia l'esecuzione... e a quel punto inizia l'esecuzione... e a quel punto inizia l'esecuzione...

+ La terminazione di un processo implica la riappropriazione da parte del SO di ogni risorsa riusabile → ad esempio per la memoria → una volta che il processo ha terminato ci si deve riappropriare della memoria che è stata utilizzata da questo processo, altrimenti dopo solo alcuni processi eseguiti la disponibilità della memoria finisce!

+Si possono avere processi *single-thread* oppure processi *multi-thread*

■ Un processo *single-thread* (quelli visti fin'ora) è un semplice programma che esegue una cosa una dopo l'altra per risolvere un qualche problema... ha quindi un singolo program counter che specifica la locazione della prossima istruzione da eseguire → Il processo esegue le istruzioni sequenzialmente, una alla volta, fino al completamento

■ Un processo *multi-thread* → dove all'interno dello stesso processo ho più sequenze! (*Fili di esecuzione*)
(Fili di esecuzione)
(Fili di esecuzione)

→ sequenze di istruzioni che faranno cose diverse!

→ ha quindi un program counter per thread e Tipicamente il sistema ha molti processi in esecuzione su una o più CPU, alcuni processi sono degli utenti, , , altri del sistema operativo
→ Si ha quindi una Concorrenza dell'uso della CPU ottenuta facendo *multiplexing* dei vari processi / threads → si affida la CPU ad un processo, poi ad un altro e così via → bloccando tutti i processi per ogni CPU in modo che riescano a proseguire...

+Quali sono le attività della gestione dei processi?

- Creare e cancellare processi utente o del sistema operativo → dar possibilità di far partire un processo! (che non è molto semplice in quanto c'è da fare tutta un'attività di caricamento del processo → ad esempio il codice dal disco dev'essere preso, allocato e in memoria, trovato le giuste zone di memoria per ottenere tutto il codice, inizializzare i dati iniziali ecc...) → anche cancellare è una parte importante, ad esempio nel caso di un processo che sta occupando il sistema inutilmente dev'essere possibilmente per l'amministratore del sistema fermare il processo!
- Sospendere e riprendere processi → far rimanere in attesa certi processi in attesa di altri processi in esecuzione —> questo però salvando lo stato in cui si trova il

processo—> questo però salvando lo stato in cui si trova il processo—> questo però salvando lo stato in cui si trova il processo

- Fornire meccanismi per la sincronizzazione dei processi → aspettare che un processo si sincronizzi con altri... ad esempio nel caso dei semafori... ad esempio nel caso dei semafori... ad esempio nel caso dei semafori
- Fornire meccanismi per la comunicazione tra processi
- Fornire meccanismi per la gestione dello stallo (deadlock) —> questo non in tutti i sistemi operativi! —> questo non in tutti i sistemi operativi! —> questo non in tutti i sistemi operativi!

+Oltre quindi alla gestione dei processi, c'è anche da gestire come questi siano portati all'interno della CPU → gestione della memoria in modo da garantire tempi di risposta non troppo lunghi... in particolare Tutte le istruzioni devono essere in memoria per poter essere eseguite → inoltre La gestione della memoria determina cosa è in memoria e quando dunque se ben utilizzata Ottimizza l'utilizzo della CPU e la risposta del sistema!

+Dunque le Attività di gestione della memoria sono:

- Tiene traccia di quali parti della memoria sono utilizzate e da chi
- Decide quali processi (o sue parti) e dati muovere in/out dalla memoria primaria
- Alloca e disalloca spazi di memoria in base alle necessità del sistema del sistema

+Altro compito importante di un sistema operativo è la gestione della memoria secondaria → file-system...(dispositivi di memorizzazione di massa)

II (dispositivi di memorizzazione di massa)

II (dispositivi di memorizzazione di massa)

Il Sistema Operativo fornisce una vista uniforme e logica della memoria secondaria → la memoria secondaria vista come dischi magnetici in realtà sarebbe accessibile come dei blocchi di byte che possono letti o scritti nell'ordine dei kilobyte, per mezzo di un accesso diretto → dunque tempo costante per accedere (anche se comunque elevato)

+é nata la necessità di avere un astrazione sopra quei livelli in modo da rendere utilizzabili in modo più semplice!

→ il file-system fà dunque da intermediario creando zone di astrazione del disco (?) che sono i file! → che sono visti dai programmi come sequenze di byte che possono essere lette o scritte sul disco —> mappare in zone di memoria! —> mappare in zone di memoria! —> mappare in zone di memoria!

+Un astrazione successiva è che i file possono essere organizzati in delle directories → le quali a loro volta possono essere messe una dentro l'altra a creare una gerarchia in modo da non avere sequenze enormi di file in cui trovare il file che ci interessa diventa molto dispesioso in termini di tempo!

→ dunque compito del SO sarà usare questa astrazione in modo da rendere più semplice sia ai programmatori delle applicazioni e poi anche agli utenti l'accesso a questo tipo di risorse!

+Attività del SO includono

- Creare e cancellare files e directories
- Primitive per manipolare files e directories
- Mappare files in memoria primaria → far vedere il contenuto del file
- Backup files su supporti di memorizzazione stabili (non volatili)

+Spostare un file da una directory ad una altra è un'operazione costosa... per cui invece di copiare il tutto si tende a cambiare in memoria una certa struttura dati che contiene i vari nomi o indirizzi delle varie directory in modo da ritrovare subito il file che cercavo in una certa directory! (ancora astrazione!)

(nel caso invece bisogna spostare il file da un dispositivo ad un altro → ad esempio una chiavetta usb → per questa operazione occorrerà più tempo!)

+Gestione memoria di massa

→ Solitamente i dischi sono usati per memorizzare dati che non entrano in memoria primaria o dati che devono essere memorizzati per lungo tempo. → gestirlo dunque in modo oculato/attento risulta fondamentale!

→ Spesso addirittura l'intera velocità del calcolatore è basata sul sottosistema di gestione dei dischi e sui suoi algoritmi di gestione! → siccome è normalmente il dispositivo più lento che utilizziamo, se abbiamo applicazioni che lo utilizzano quelle saranno soggette a problemi di performance! (ad esempio per sistemi database → dovrà avere dei sottosistemi che dovranno quindi essere gestiti in modo opportuno!)

+Attività del sistema operativo per quanto riguarda la memoria di massa sono:

- Gestione dello spazio libero

- Allocazione → dei file e delle directory sul disco → di nuovo il mapping delle zone di memoria nella memoria principale → di nuovo il mapping delle zone di memoria nella memoria principale → di nuovo il mapping delle zone di memoria nella memoria principale

- Scheduling del disco → decidere come posizionare il lettore della testina in modo tale da minimizzare il tempo di accesso alla prossima locazione su cui vogliamo andare!

→ se ad esempio dobbiamo fare riferimento ad un settore che è immediatamente vicino → in cui basta spostare la testina di poco e siamo subito nel settore che ci interessa allora risulta conveniente fare questo passaggio piuttosto che andare a leggere andare a leggere un altro dato che si trova in un'altra zona di memoria che è più lontana e dunque comporta più tempo! → il costo di lettura del disco non è sempre uniforme (come nel caso della memoria primaria) → infatti comporta un movimento fisico della testina che dunque comporta un ritardo che può essere variabile a seconda della posizione in cui ci si trova il dato da ricercare!

→ algoritmi di scheduling del disco tengono conto di questo fatto per decidere come far avvenire al disco l'accesso in modo più veloce!

Ad oggi invece con gli ssd si è parzialmente risolto il problema in quanto l'accesso in memoria è uniforme!

Sottosistema I/O

Ad oggi invece con gli ssd si è parzialmente risolto il problema in quanto l'accesso in memoria è uniforme!

Sottosistema I/O

Ad oggi invece con gli ssd si è parzialmente risolto il problema in quanto l'accesso in memoria è uniforme!

Sottosistema I/O

Uno degli scopi dei Sistemi Operativi è di nascondere all'utente le peculiarità dei dispositivi hardware!

→ Il sottosistema I/O è responsabile di:

- Gestione della memoria per I/O ovvero per tutte quelle attività di:

→ *buffering* (memorizzare i dati temporaneamente mentre vengono trasferiti) → una parte della memoria verrà usata dal sistema operativo per gestire ad esempio il trasferimento di dati → ad esempio per copiare un file da un dispositivo ad un altro, questo prima verrà messo sulla memoria primaria e successivamente sul nuovo dispositivo

→ *caching* (memorizzare dati da memorie più lente (ad esempio dischi) in memorie veloci per aumentare la performance dell'accesso a questi dati),

→ *spooling* (memorizzazione dati verso device lenti, I/O asincrono) → ad esempio per una stampa → i dati vengono messi da una parte in coda e poi piano piano i dati vengono mandati al dispositivo in modo tale da produrre la richiesta voluta

→ se tanti processi vorranno fare la stampa questi si metteranno in coda e in modo asincrono vengono mandati i dati al dispositivo

- Interfaccia di uso generale tra device e driver → parte di sistema operativo che si occupa di gestire il dispositivo fisico che è appunto collegato!

- Drivers per dispositivi speciali

Protezione e sicurezza

→ Un altro compito dei sistemi operativi è quello di gestire la protezione e la sicurezza del sistema → con *protezione* si intende quindi di attivare una serie di meccanismi per controllare gli accessi a dati e processi da parte degli utenti!

→ si deve quindi definire a quali cose un utente può accedere e a quali no → ad esempio il filesystem si occupa di dire a quali file l'utente può accedere e a quali no! → oppure ancora nel caso di un sistema condiviso con vari utenti, gli altri non devono vedere i nostri file memorizzati sul mio sistema... a meno che non diamo la possibilità di accedere alle nostre risorse anche ad un altro utente

+Anche le applicazioni dovranno garantire la protezione se offrono dei servizi verso altri sistemi (ovviamente non c'è solo il sistema operativo a gestire la protezione dei dati! → ad esempio per applicazioni web → l'accesso protetto alla nostra applicazione dovrà essere gestita personalmente!) → pensando alle applicazioni per i dispositivi mobile che gestiscono l'autenticazione tramite "biometria" → ad esempio con la firma digitale oppure il riconoscimento del viso → questo è un servizio di protezione che viene fornito dal sistema operativo per permettere il riconoscimento dell'utente!

+Poi sono presenti aspetti di sicurezza da attacchi che possono essere o interni o esterni

- Tipo: denial-of-service, worms, viruses, furto di identità, furto di servizi

→ compito quindi del sistema operativo è poter dare/fornire dei servizi che aiutano a identificare/capire se abbiamo avuto questo tipo di problemi

+Tipicamente dunque i sistemi operativi forniscono un astrazione che è una rappresentazione interna dell'utente che permettono di gestire l'identità degli utenti → *user ID* → i quali associano una serie di informazioni a questa astrazione che è il dato rappresentato all'interno del calcolatore → questo User ID è associato a tutti i file e processi dell'utente per realizzare il controllo degli accessi

+Per facilitare l'assegnazione di questi permessi si possono avere dei gruppi identificativi → nel caso ad esempio esistano migliaia di utenti, diventa più utile dividere in gruppi i vari utenti secondo determinate caratteristiche → raggruppo dunque gli utenti in un certo gruppo e ne condivido i permessi solo a questo determinato gruppo → in modo tale che un nuovo arrivo di un certo utente, mi basta aggiungerlo a gruppo e dargli i nuovi permessi

+Esiste infine la possibilità di *privilege escalation* → permettere all'utente di cambiare il proprio ID per avere più diritti → ovvero per diventare un super user oppure il *root* → di modo che possa avere accesso ad un livello di sicurezza più importante

Ambienti di elaborazione(partie inutile)

■ Personal Computer

- In ufficio

→ PCs connessi a una rete locale, terminali connessi a un mainframe o minicomputers forniscono accessi batch e in timesharing

→ Portali permettono a sistemi in rete e remoti l'accesso alle stesse risorse

- A casa

→ Era un singolo sistema

→ Ora più sistemi in rete dietro firewall → in sostanza il nostro wifi di casa (che all'inizio era una sorta di barriera esterna → non esisteva il wifi libero)

- Client-Server Computing

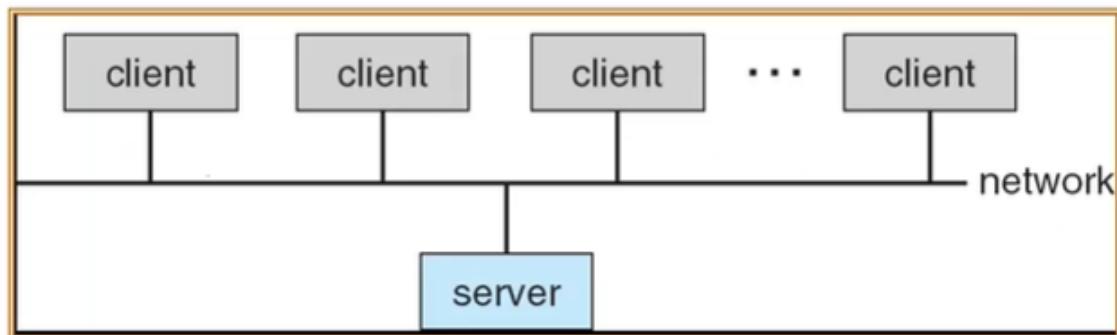
→ una serie di client connessi ad un server, il quale permette certi servizi ad esempio:

→ *Compute-server* fornisce una interfaccia ai client per richiedere servizi (es. database) → offrendo dei servizi di memorizzazione/query ai client

→ *File-server* fornisce una interfaccia ai client per memorizzare e accedere ai file (nel server centrale vengono memorizzati i dati) → i client sono dei pc “normali”

→ il file-server dà quindi la possibilità di condividere più dati dei client utilizzando un unico server → che permette di accedere in modo sicuro ai file che sono utilizzabili

→ dunque in sostanza abbiamo un unico server che da accesso ai client stupidi? sudditi?



+Sarà compito del sistema operativo darà un astrazione ulteriore che permetta di accedere a qualcosa che si pensa sia il contenuto di un disco ma in realtà non è lì attaccato sul client ma è attaccato da un'altra parte → il SO fa come al solito da intermediario alla rete, permettendo di far accedere ai file su un determinato client ai dati che sono nel server

→ cosa succede però se due client vogliono accedere contemporaneamente ad uno stesso file ma questo viene modificato prima da uno e poi dall'altro → questo comporta problemi di

sincronizzazione di gestione di accesso a questi dati condivisi che sono distribuiti nel sistema

→ l'altra possibilità è il peer-to-peer → dove non si distingue fra server e client, ma tutti i nodi sono considerati alla pari (peers) → saranno servers per alcuni aspetti e client per altri... Un nodo deve connettersi a una rete P2P → può essere un servizio centralizzato che tiene traccia di quali sono i servizi e a chi li offre

→ nel caso P2P abbiamo una comunicazione anche diretta fra i client (o peers), è presente comunque un server centrale nel quale viene tenuto lo storico di quali sono i servizi esposti dai vari peers

+Esistono anche casi in cui non è presente un servizio centralizzato, dunque si sostituisce questa funzione facendo delle richieste in broadcast (rete locale) nel quale si richiede l'elenco dei servizi offerti da ogni client e quel punto si realizza il protocollo di discovery → di scoperta → una scoperti i vari servizi offerti dai vari peer allora la comunicazione sarà diretta
+esempi sono BitTorrent, Gnutella ecc...

Il resto di ambienti di elaborazione vedi slide se vuoi

Struttura dei sistemi operativi

→ quali sono i servizi principali di un sistema operativo?



→ per accounting si intende il servizio che tiene traccia di chi usa cosa → usato spesso per quando si usa un sistema condiviso

+Dunque l'insieme dei servizi offerti dal sistema operativo fornisce funzioni utili per l'utente:

- **Interfaccia Utente** (User interface) – Quasi tutti i sistemi operativi hanno una interfaccia utente (UI) → la quale varia tra Command-Line (CLI) → tipico ad esempio degli ambienti linux, oppure Graphics User Interface (GUI) → che sono ambienti grafici per l'interazione in modo più naturale del sistema, Batch
- **Esecuzione dei programmi** – Il sistema deve poter caricare un programma in memoria ed eseguirlo, terminare la sua esecuzione, sia normale che in caso di errore

- **Operazioni di I/O** – Un programma in esecuzione può aver bisogno di I/O per l'accesso a file o per interagire con un dispositivo. → dunque la gestione di mouse/tastiera...
- **Accesso a File-system** – Il File-system è di particolare interesse. I programmi hanno bisogno di leggere/scrivere *file* e directories, crearle e cancellarle, ricercarle, etc

+Altri aspetti sono:

- **Comunicazioni** – I processi possono scambiare informazioni, su uno stesso calcolatore o tra calcolatori in rete → La comunicazione può essere fatta attraverso memoria condivisa o attraverso invio di messaggi (pacchetti inviati dal SO) → nel caso di memoria condivisa lo si fa in genere su uno stesso calcolatore...
 - **Rilevazione errori** – Il SO deve essere sempre attento a possibili errori!
 - Possono avvenire nella CPU e in memoria, in dispositivi I/O e in programmi utenti
 - Per ogni tipo di errore il SO dovrebbe effettuare l'azione appropriata per garantire una elaborazione corretta e consistente → deve impedire ad esempio che un processo possa andare ad usare della memoria che non lo riguarda → in quel caso deve terminare il processo o comunque impedire che questo accada!
- +Fornisce infine un supporto per il **debug** delle applicazioni che può migliorare notevolmente la possibilità di usare efficientemente il sistema da parte dell'utente e del programmatore → riuscendo a trovare gli errori in modo più semplice!

+oppure ancora:

- **Allocazione risorse** – Quando più utenti o più job sono in esecuzione contemporaneamente, ognuno deve avere a disposizione delle risorse
→ Abbiamo molti tipi di risorse → Alcuni come cicli di CPU(tempo di cpu usato per l'allocazione del programma), spazi di memoria primaria, e file storage possono (anzi devono) avere una allocazione specifica, altri come dispositivi di I/O possono avere una gestione generale di tipo richiesta/rilascio → ad esempio per la stampante... posso fare un stampa e poi passare ad altro
- **Accounting/Logging** – (di nuovo) Per tenere traccia di quali utenti usano, quanto e quali tipi di risorse di sistema → ad esempio nel caso il sistema venga compromesso avere un sistema di logging che tiene traccia di cosa l'utente o quell'applicazione ha fatto... permette di capire meglio se l'attacco è stato portato a buon fine e permette di capire a cosa ha avuto accesso in modo non sicuro
- **Protezione e sicurezza** – I possessori delle informazioni memorizzate in un sistema multiutente o in rete possono voler controllare l'uso delle informazioni, processi concorrenti non dovrebbero interferire tra loro
→ Protezione assicura che tutti gli accessi al sistema sono controllati
→ Sicurezza del sistema dall'esterno: richiede l'uso dell'autenticazione degli utenti, si estende alla difesa di dispositivi di I/O esterni dai tentativi di uso non valido → in modo da permettere un accesso sicuro!
→ Se un sistema deve essere protetto e sicuro, delle precauzioni devo essere prese in ogni sua parte. Una catena è forte come il suo anello più debole.
→ tipicamente l'anello più debole è l'utente!

Interfacce utente del sistema operativo

- + **Command Line Interface** → alcune volte implementato nel kernel, altre nei programmi di sistema → dunque un interfaccia nella quale vengono forniti una serie di comandi...
→ In alcuni sistemi sono forniti anche più interfacce a comandi → le così dette *shells* → le quali sono programmabili oppure posso realizzare una file che contengono un insieme di istruzioni → che dunque facilitano una serie di attività le quali devono essere fatte in modo regolare → senza dovermi ricordare di tutti i comandi! → eseguo quindi una sorta di procedura che tiene tutti questi comandi e li esegue.
- + Dunque l'interfaccia fondamentalmente riceve un comando dall'utente e lo esegue
→ Alcune volte i comandi sono built-in nell'interprete dei comandi, altre volte sono solo nomi dei programmi, che vengono caricati e poi eseguiti → in questo caso aggiungere nuovi comandi non richiede di modificare la shell
→ se invece abbiamo un interfaccia in cui una riga di comando esegue solo dei comandi presi dalla shell diventa necessario estenderlo

- + **Graphic User Interface (GUI)** → è Interfaccia user-friendly che usa la metafora della scrivania (desktop) → si basa su mouse, tastiera, e monitor...
→ menter le Icone rappresentano file, programmi, azioni, ecc.
→ I pulsanti del mouse premuti su oggetti dell'interfaccia provocano varie azioni (danno informazioni, aprono opzioni, eseguono funzioni, aprono directory/cartelle)
- + Questa modalità di interazione è stata inventata negli anni 70 al Xerox PARC (1973)
+ Molti sistemi includono interfacce CLI e GUI, ovvero entrambi i tipi di interfacce:

- Microsoft Windows ha GUI con CLI “command” shell
- Apple Mac OS X ha “Aqua” GUI con sotto un kernel UNIX e sono disponibili le shell unix
- Solaris ha CLI e delle GUI (Java Desktop, KDE) → sistema operativo sviluppato non in tempi molto recenti...

Chiamate di sistema

→ Una delle parti fondamentali di un sistema operativo è quella di realizzare le **chiamate di sistema** → cosa sono??

Sono delle Interfacce di programmazione verso i *servizi forniti* dal Sistema operativo e Tipicamente sono scritte in C/C++

→ Principalmente usate dai programmi attraverso Application Program Interface (API) che forniscono un’astrazione di alto livello per l’accesso alla funzionalità offerta dal sistema operativo → in realtà quella che utilizziamo come chiamata di sistema, in un programma C la vediamo come una funzione(che prende certi argomenti e restituisce un certo risultato)

→ c’è dunque la chiamata del sistema fatta usando interruzioni o istruzioni particolari che permettono di passare dalla modalità utente alla modalità di sistema!

+Le tre API più comuni sono: Win32 API per Windows, POSIX API per sistemi

POSIX-based (includendo tutte le versioni di UNIX, Linux, e Mac OS X), e Java API per la Java virtual machine (JVM) → in questo caso questi sono ad un livello di astrazione superiore → è infatti la JVM che fa le chiamate di sistema opportune per il sistema operativo per il quale è in esecuzione!

+Un esempio di chiamate di sistema → è l’operazione di trasferire un contenuto da un file di origine a quello di un file di destinazione

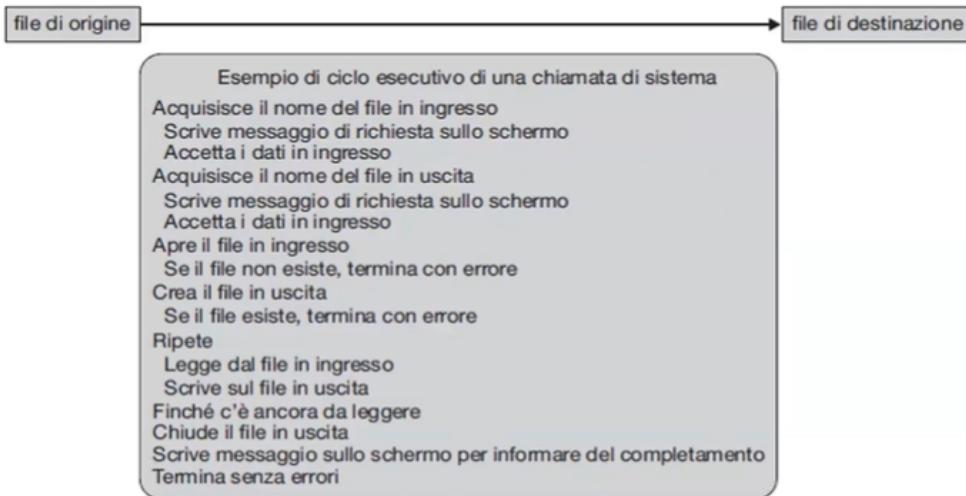
→ il programma dunque che fa questo dovrà prima acquisire il nome del file in ingresso che porterà alla scrittura di un messaggio di richiesta sullo schermo → e per fare questo farà una prima chiamata di sistema!

→ si accettano quindi i dati in ingresso → e si acquisisce il nome del file d’uscita, cioè scrivo un altro messaggio per chiedere: “ dimmi qual è il nome del file su cui vuoi scrivere...”

→ poi si metterà in attesa dei dati dell’input dell’utente e anche questo lo farà usando un’altra chiamata di sistema!

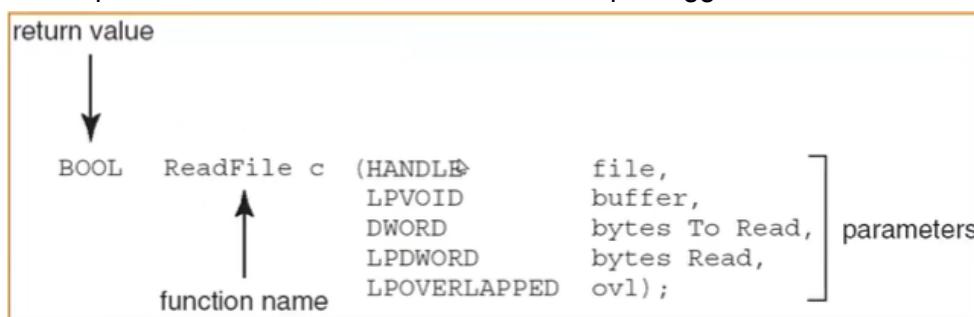
→ Una volta raccolti i dati necessari, per prima cosa dovrà aprire il file in ingresso → e anche stavolta userà una chiamata di sistema per aprire il file → se però il file non esiste dovrà far terminare il processo generando un errore!

→ anche la terminazione del processo sarà un’chiamata del sistema!



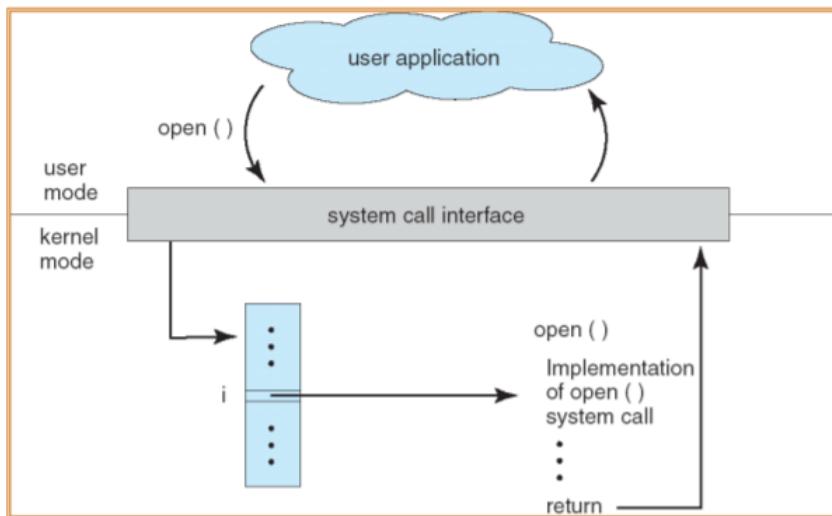
- tutte le attività che hanno riferimento all'esterno del programma → ovvero ciò che non è computazione diretta del programma → dunque manipolazione dei dati, che si trova in qualche modo in memoria comporta una chiamata di sistema per interagire con il sistema operativo (in questo caso con l'hardware)
- esistono comunque anche chiamate di sistema che servono per interagire proprie del sistema operativo che non sono dell'hardware!
- ad esempio le chiamate di sistema che servono a gestire i semafori! → i quali non sono dispositivi fisici → ma astrazioni create dal sistema operativo e che sono utilizzabili tramite una chiamata di sistema !

+Esempio → chiamata di sistema Win32 API - per leggere dei dati da un file (*ReadFile()*)



- un parametro di tipo HANDLE è un identificatore del file (che ho aperto precedentemente)
- prima ho usato un API per avere l'HANDLE del file → ovvero ho tentato un apertura del file per leggerlo → a quel punto uso un readfile per leggermi da questo file, attraverso un puntatore a una zona di memoria dove mettere i dati (buffer) → poi gli dico "guarda devi leggermi tot dati" al massimo → ovvero fisso una capacità massima del buffer
- aggiungo poi un puntatore ad una double-word in modo da dire quanti effettivamente ne ha letti → "io ti ho chiesto di leggere X byte, ma in realtà ne ho letti Y perchè non ce n'erano abbastanza..."
- +Infine è presente il parametro overlapped → che serve per fare un I/O asincrono → nel momento in cui la chiamata termina → allora nel buffer ci sono i byte che devono essere letti → quindi la struttura dati mi permette di capire quando questa cosa è fatta o meno

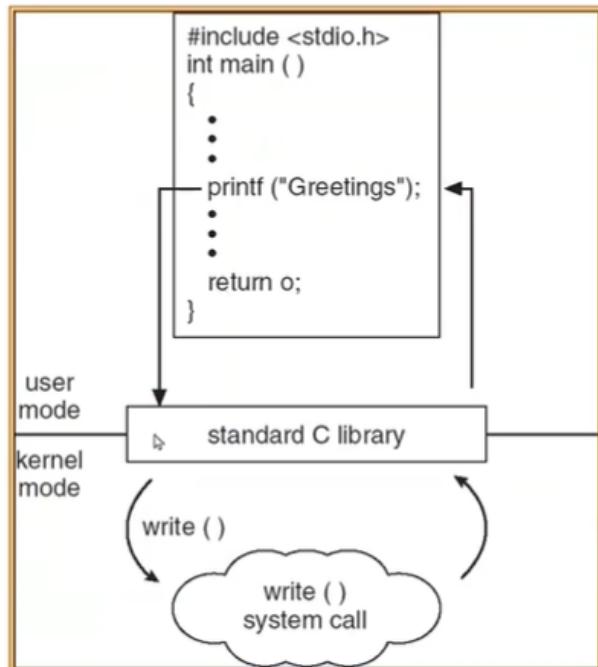
- + Tipicamente ogni systemCall è identificata da un numero → nell'implementazione del kernel, abbiamo l'identificazione tramite un numero → il sistema operativo si tiene una tabella nella quale ad ogni numero corrisponde una procedura da eseguire!
- I passaggi sono quindi... L'interfaccia delle chiamate di sistema invoca la chiamata di sistema a **livello di kernel** e ritorna lo stato della chiamata e ogni valore di ritorno!
- + Il chiamante non ha necessità di sapere come la chiamata di sistema è implementata → deve solo eseguire l'API e capire cosa il sistema il sistema operativo farà come risultato della chiamata (tipicamente un bravo programmatore si legge la documentazione dell'API, cerca di capire quali sono le sue funzionalità e come realizzare i vari scopi!)
- + Infine i dettagli sono nascosti al programmatore proprio dall'API!!
- abbiamo infine una gestione di librerie a run-time che permettono di usare queste API.



- l'utente usa l'interfaccia verso il system call → che poi farà l'effettiva chiamata alla procedura in kernel mode → e questa chiamata la farà usando un numero che utilizza come indice di una tabella che il sistema operativo si tiene!
- alla posizione "i" → ci sarà un puntatore alla funzione open() che realizza appunto l'implementazione della chiamata di sistema di apertura di un file
- una volta terminata l'esecuzione della procedura si esce dall'interfaccia system call e verrà chiamata la funzione che per l'applicazione è banalmente una funzione → ma in realtà dietro l'esecuzione di questa c'è una chiamata di sistema che viene implementata in questo modo → ovvero passa alla modalità kernel, esegue il codice e poi ritorna...

- + Un altro esempio, sta volta con il C → quando usiamo una printf() → questa è una funzione di libreria → la quale una volta costruita la sequenza che deve essere stampata → utilizza la chiamata di sistema write() per scriverlo nello standard output del processo!
- la chiamata di sistema porterà all'effettiva stampa della stringa → e quando termina la chiamata si riporterà alla terminazione della printf()
- abbiamo quindi 2 livelli → la prima riguarda la gestione di livello del linguaggio eseguito in modalità utente e poi invece la chiamata di sistema che effettua veramente l'operazione è in realtà realizzata in modalità kernel del sistema operativo
- + printf che in realtà ci dà un livello di astrazione superiore → aiuta dunque a dare un output più semplice...

Api < -- > per dividere l'interfaccia in più parti!



→ notare dunque che in questo caso ci sono 2 livelli di astrazione → 1 che permette di accedere facilmente al buffer → attraverso una serie di parametri → e l'astrazione successiva (fatta dalla printf()) che facilità l'utilizzo della stampa!

12/03/2021

+Esempio chiamate di sistema di linux

→ Le system call sono chiamate tramite istruzione (in forma esadecimale) INT 80h (interrupt sincrona chiamata esplicitamente) → nei processori a 32 e a 64 bit si usano delle istruzioni più specifiche del processore... ad esempio sysenter/sysexit (per 32) e syscall/sysret (per 64), che sono migliori rispetto alle semplici interruzioni...

→ dunque riprendendo nei vettori di interruzione alla specifica posizione di 80h (esadecimale) vi è associata una procedura → la quale gestisce le chiamate di sistema... dunque quando viene eseguita l'istruzione INT 80h, il processore va nella posizione indicata ed esegue la procedura → e nel fare questa cosa qui passa dalla modalità da utente a sistema!

→ esiste una sola posizione nell'array di interruzione che serve a gestire le chiamate di sistema e in più il registro EAX contiene il numero della system call da eseguire, infine gli altri registri conterranno altri parametri necessari alla system call → ogni chiamata di sistema ha i suoi parametri e utilizza i vari registri che sono a disposizione

+per un elenco delle system call si veda

<https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md> !

+Continuando con i sistemi linux...

Il kernel nella procedura associata all' interruzione 80h (ISR) può controllare se la chiamata è valida e usando una tabella interna al kernel può chiamare la funzione associata alla syscall indicata da registro EAX.

+ Prima cosa dunque si andrà a controllare che per una determinata procedura ci siano i parametri giusti → al termine quindi in EAX è presente il risultato, nel quale verrà messo <0 in caso di fallimento, oppure >=0 per successo
→ ad esempio se volessimo fare una chiamata di sistema *write()* per scrivere sullo standard output:

MOV EAX, 04h	// write syscall
MOV EBX, 1	// standard output
MOV ECX, buffer	// puntatore al buffer di caratteri da stampare
MOV EDX, count	// numero di caratteri del buffer da stampare
INT 80h	// esegue la syscall

→ notare che la int 80h venga chiamata solo dopo aver settato tutti i parametri necessari...
→ successivamente si controlla se EAX contiene un valore minore o uguale/maggiore a zero... e in base a quello decidere cosa fare

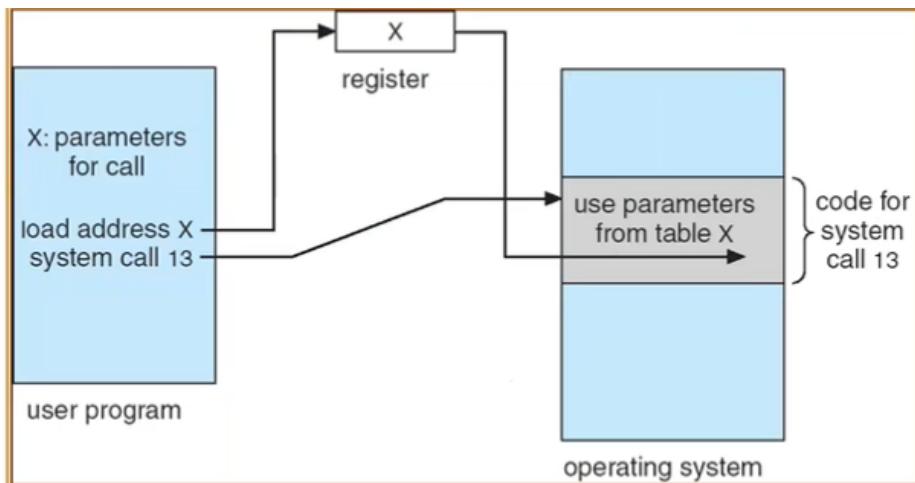
+ Per facilitare queste chiamate di sistema sono state definite particolari funzioni che una volta che sono state definite e che hanno preso dei parametri, in base alla chiamate di sistema immettono il numero giusto nei registri i valori che devono essere messi in base alla specifica della chiamata di sistema
→ l'API *write* sarà una funzione che praticamente prende come argomento 3 parametri (i valori da inserire nei registri) e a quel punto fa l'int... infine guarderà il risultato in un determinato registro e lo restituisce

+ La *printf* non è una chiamata di sistema → ma è un semplice funzione del C che come abbiamo visto l'altra volta va a usare la chiamata di sistema *write* appunto per stampare sullo schermo una stringa (funzione disponibile con il linguaggio! → dunque sempre un API ma di più alto livello rispetto alle chiamate di sistema)

Passaggio di parametri fra chiamate di sistema

Esistono Tre metodi generali che sono usati per passare parametri a SO:

- Il piu' semplice: passare i parametri nei registri della CPU !
→ In alcuni casi però si possono avere piu' parametri che registri! → in quel caso avrò dei problemi → Usato da Linux
 - Parametri memorizzati in un blocco o tabella in memoria, dunque nel momento in cui dobbiamo eseguire una certa chiamata di sistema → l'indirizzo del blocco viene passato come parametro in un registro → è un solo parametro che indirizza una zona di memoria dove sono presenti altri parametri → Usato da Solaris
 - Parametri pushed sullo **stack** dal programma e popped dallo stack dal SO → il programma immette i parametri sullo stack, mentre il pop viene eseguito dal sistema operativo!
→ Metodi che usano il blocco e lo stack non limitano il numero e la lunghezza dei parametri passati.



→ esempio di come avviene il passaggio di parametri con la memoria → un certo registro sarà usato per indicare la zona di memoria dove sono presenti i parametri per la chiamata di sistema che si vuole eseguire

+Le principali chiamate di sistema sono quindi:

→ Controllo processi ■ Gestione File ■ Gestione Dispositivi ■ Comunicazioni

Programmi di sistema

I programmi di sistema → che sono forniti insieme al sistema operativo → forniscono un ambiente per lo sviluppo di programmi e per la loro esecuzione.

→ Possono essere divisi in programmi per:

- La manipolazione di file
- Ottenere informazioni sullo stato
- La modifica di file
- Il supporto di linguaggi di programmazione
- Il caricamento di programmi e loro esecuzione
- Le comunicazioni
- Programmi applicativi

+ Per la maggior parte degli utenti un sistema operativo è definito dai programmi di sistema disponibili, non dalle chiamate di sistema fornite. → in quanto gli utenti usano le chiamate di sistema per interagire con il sistema e fare le cose minime che sono necessarie nell'uso comune del sistema !

+Dunque Forniscono un ambiente per lo sviluppo ed esecuzione dei programmi → alcuni sono semplicemente delle interfacce per le chiamate di sistema; altri sono considerevolmente più complessi.

+La parte di **gestione file** → permette di Creare, cancellare, copiare, rinominare, stampare, generalmente manipola file e directory del sistema.

+Permette anche **Informazione sullo stato** → ambiente che fornisce informazioni sullo stato del sistema:

- Alcuni chiedono al sistema informazioni – data e ora, quantità di memoria disponibile, uso spazio disco, numero di utenti
- Altri forniscono informazioni dettagliate per l'analisi della performance, per il logging, e per il debug → le quali permettono appunto di vedere cosa sta succedendo al sistema e quindi vedere se sono presenti problemi oppure quali applicazioni usano troppa CPU rallentando il sistema ecc...

- Tipicamente questi programmi stampano l'output su terminale o altri dispositivi di output
- + Alcuni sistemi implementano un registro → posizione centralizzata → usato per memorizzare e reperire informazioni sulla configurazione e l'esecuzione delle applicazioni.
- +Oppure ancora dei programmi per la Modifica di file → ad esempio editor di testo per creare e modificare file oppure ancora comandi speciali per cercare nei file o per effettuare trasformazioni del testo
- +In alcuni linguaggi sono presenti supporti per i linguaggi di programmazione (ad esempio i sistemi nei unix) – Compilatori, assemblatori, debugger e interpreti
- +Programmi per il caricamento ed esecuzione → Absolute loaders, relocatable loaders → che sono programmi per il caricamento in memoria, linkage editors, e overlayloaders, programmi per il debug per linguaggi di alto livello e per linguaggio macchina
- +Programmi per la Comunicazione → Fornisce meccanismi per creare connessioni virtuali tra processi, utenti e calcolatori → Permettere quindi agli utenti di scambiarsi messaggi, navigare pagine web, inviare e-mail, collegarsi in modo remoto, trasferire file tra macchine diverse → ne sono un esempio i browser ecc...

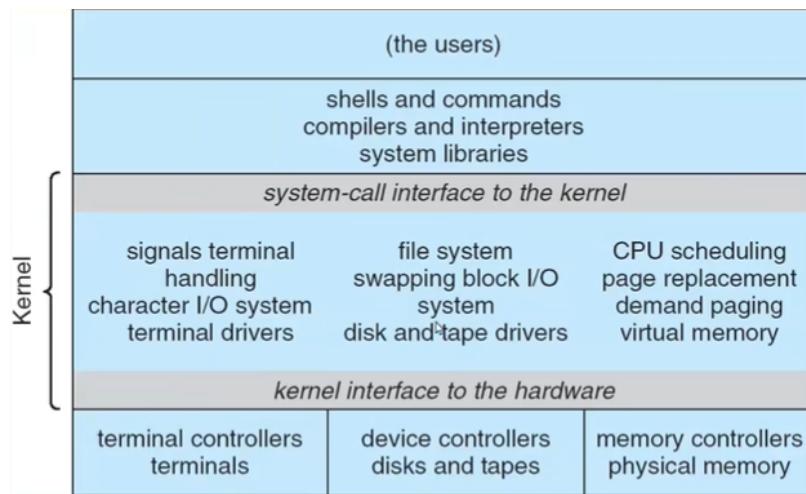
Progettazione e Implementazione di sistemi operativi

- Progettare e implementare un sistema operativo è un problema difficile per molti aspetti... Ad esempio la struttura interna di sistemi operativi può variare moltissimo!
- +Si inizia dunque definendo gli obiettivi (i goals) e le caratteristiche del sistema → molto dipende dall'hardware per il quale deve essere sviluppato questo sistema e quindi dal tipo di sistema che si vuole realizzare (a lotti, in time-sharing, mono/multiutente, realtime o uso generale) → si hanno quindi una serie di user-goal e sistem-goal:
 - User goals – il sistema operativo dovrebbe essere facile da usare, facile da imparare, affidabile, sicuro e veloce
 - System goals – il sistema operativo dovrebbe essere facile da progettare, implementare e manutenere, così come essere flessibile, affidabile e efficiente
- +Una cosa importante è riuscire a separare *criteri o politiche* → ovvero cosa sarà fatto... dai *meccanismi* → ovvero come sarà fatto → come certe cose saranno implementate
 - sarà dunque il meccanismo a determinare come fare qualcosa mentre i criteri ci dicono cosa implementare
 - La separazione tra criteri e meccanismi è un principio molto importante permette la massima flessibilità se i criteri cambiano nel tempo!
 - ad esempio il TIMER è un meccanismo, mentre la decisione di quanto tempo assegnare ad un processo riguarda un criterio! → ad esempio la politica di *scheduling* → la quale a sua volta si basa sul meccanismo del timer!
 - per l'assegnazione delle risorse sono importanti i criteri

+Struttura MONOLITICA

- approccio più semplice in cui il kernel è un unico file binario ed ha un unico spazio di indirizzamento, usato dal kernel
- al boot viene caricato e quindi messo in esecuzione...
- è questa una struttura che fù inizialmente usata da sistemi unix e linux... anche se gli approcci nuovi non cambiano molto da questo
- +Il problema di questo approccio è che i *device driver* sono integrati nel kernel dunque diventa difficile supportare nuovi device → in quanto implica ricompilare tutto il kernel e fare il reboot del nuovo sistema! → se aggiungo un nuovo device non è tanto agevole dover sia ricompilare il kernel che fare successivamente il boot...

+Struttura UNIX



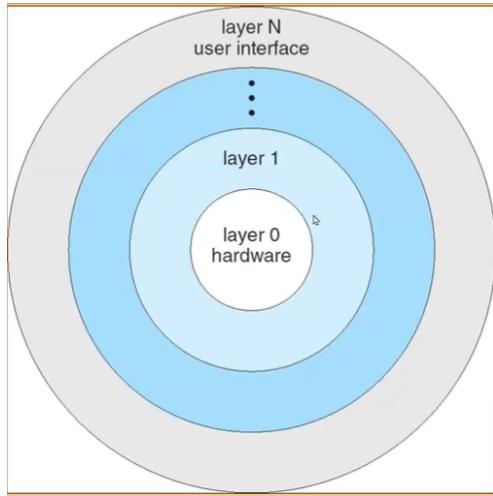
→ notare che il kernel è monolitico → è quindi un singolo programma in esecuzione!

Approccio stratificato

→ per cercare di ovviare a questo tipo di problemi è stato pensato un approccio stratificato
 → nel quale il sistema operativo è diviso in una serie di strati → ognuno costruito sulla base degli strati inferiori → Lo stato più basso (layer 0), è l'hardware, mentre lo stato più alto (layer N) è la interfaccia utente.

→ gli strati sono selezionati in modo che ognuno usa funzioni e servizi dei soli strati di livello inferiore → il problema sarà dunque definire questi in quanto ci sono dipendenze che sono difficile da staccare fra loro → pensiamo ad esempio a gestione della memoria e gestione del file system → in alcuni casi uno sta sopra e uno sta sotto... ad esempio il file system userà la memoria per fare il caching ad esempio, però poi abbiamo il problema della memoria virtuale → dove viene usato il disco come backing store → e quindi dovrebbe stare sotto! → non si risulta quindi semplice stabilire le varie posizioni...

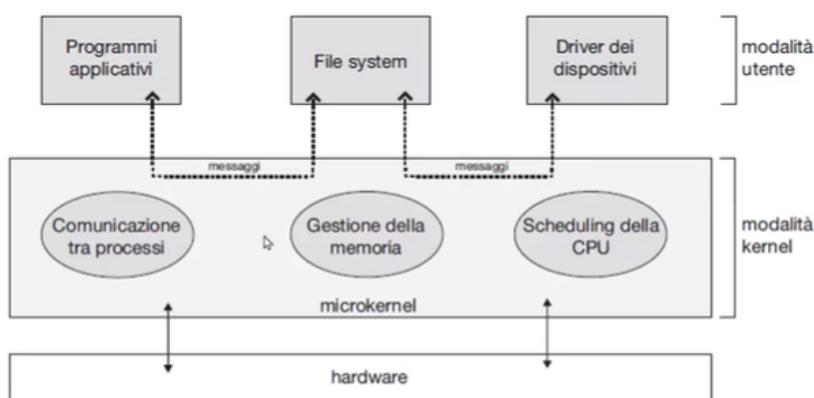
+Esempio:



- si possono dunque introdurre delle inefficienze dovute all'attraversamento dei vari strati
- che dovranno essere attraversate da varie chiamate di sistema!
- ad esempio solo arrivati ad un certo layer si riesce ad implementare una determinata richiesta! (come nel caso della memoria virtuale)

Struttura a microkernel

- +In caso di fallimento, quando ho un device driver in esecuzione in modalità kernel... se questo ha un problema, il fallimento fatto in modalità kernel può portare ad una fallimento generale del sistema! (ad esempio su windows il blue screen!)
- l'approccio opposto è quello che viene fatto nei sistemi a microkernel → dove si cerca di limitare il più possibile l'utilizzo del kernel
- il kernel si deve occupare solo delle cose fondamentali → **gestione processi, gestione della memoria, gestione della scheduling della CPU** e infine gestire la comunicazione fra processi (ovvero ciò che permette ai programmi di comunicare fra loro!)... tutto il resto che non riguarda queste cose di base viene implementato in modalità utente! → ovvero i vari processi dedicati ai programmi applicativi, i file system e infine il driver dei dispositivi verranno eseguiti in modalità utente! → questi comunicano fra loro però tramite messaggi che devono passare dalla modalità kernel



- siccome dunque ho un kernel con poco codice rispetto ai macrokernel... è meno probabile dei fallimenti del sistema(?)

→ la parte invece di gestione di device varia da sistema a sistema e può essere il tutto più problematico da implementare!

→ la parte importante è che invece in caso di fallimento del sistema viene passato al kernel e quella parte lì caso mai fallisce, ma sarà più facile da gestire!

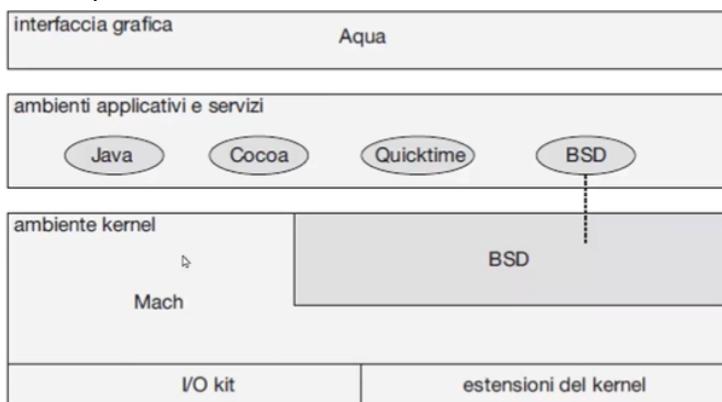
+Per fare qualsiasi cosa però ci sarà il problema che si debba passare da un certo scambio di messaggi! → ad esempio per trasferire un file molto grosso dunque da dividere in vari pezzettini ognuno dei quali necessita di una certa chiamata di sistema → ad ogni passaggio necessita di un certo messaggio per passare dalla modalità kernel alla successiva parte della modalità utente → programmi app → file system → driver dei dispositivi... e ciò introduce **una serie di attese** dovute ad aspettare che arrivino i vari messaggi! → creando vari ritardi e rendendo il sistema più inefficiente... dunque si è abbandonato l'approccio puro al microkernel

+Riassumendo...

■ Benefici:

- Più facile da estendere
 - Più facile portare il sistema operativo su nuove architetture hardware → dovrà fare il *porting* infatti di soli alcuni aspetti! → ovvero ciò che sta in modalità kernel!
 - Più affidabile → meno codice in esecuzione in modo kernel
 - Più sicuro → codice che quindi potrà essere testato molte volte!
- Demeriti:
- Diminuzione di performance dovuto alla comunicazione tra spazio utente e kernel!

+Esempio di macOS



→ *mach* è il microkernel sul quale hanno aggiunto un kernel unix BSD che fornisce una serie di chiamate di sistema unix per l'uso di applicazioni unix → diversamente non le 2 parti colloquiano direttamente, per cui non ci saranno da fare troppi passaggi fra modalità kernel e utente

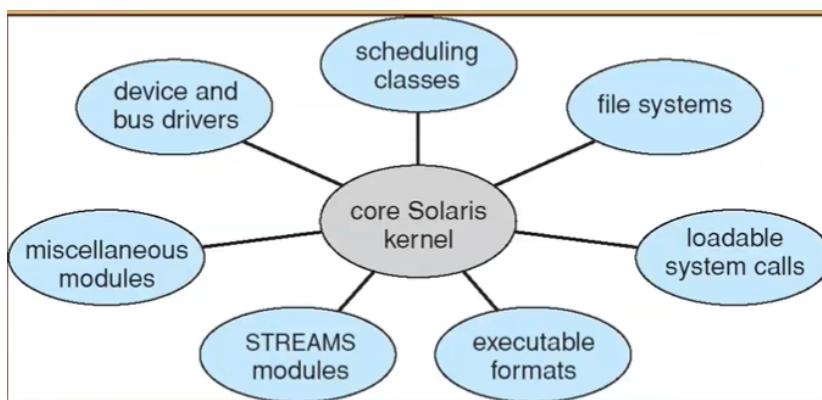
+in realtà questo è un kernel ibrido che fornisce sia aspetti micro che macro-kernel!
(gestione di I/O gestita appunto in modalità kernel)

Approccio modulare

- si fa in modo di dividere il sistema operativo in moduli che si occupano di aspetti diversi di un sistema → è quindi più un approccio object oriented → ci sarà un modulo che gestisce la cpu, altri qualcos'altro ecc...
- poi anche i singoli device driver saranno dei moduli, i quali a loro volta comunicheranno con altri moduli attraverso delle interfacce ben definite → la caratteristica è che questi moduli sono caricabili a run time (non tutti insieme come nel kernel monolitico) → dunque caricati in base alle necessità!
- approccio simile alla stratificazione ma senza una gerarchia così stretta... volendo si possono avere legami multipli in cui si ha una dipendenza di tipo specifica tra 2 moduli...
- inoltre risulta più facile aggiungere nuovi moduli!

+Ad oggi molti sistemi operativi usano un approccio modulare il quale ha un organizzazione e interfacce ben definite e in più per non ridurre la performance i device driver sono eseguiti in modalità kernel

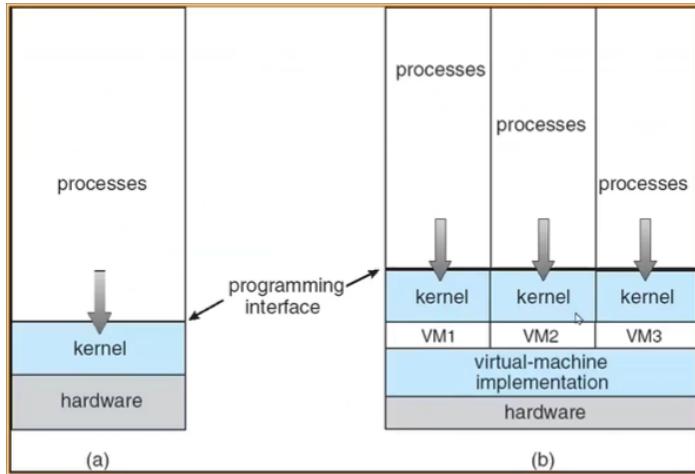
→ solaris e linux lo usano!



+anche windows 10 in realtà è considerato un kernel ibrido in quanto ha al suo interno windows NT → il quale partì come un microkernel puro ma per problemi di performance hanno aumentato le possibilità offerte dal kernel aggiungendo la possibilità di eseguire i device driver in modalità kernel!

Macchine virtuali

- una macchina virtuale è un modo con il quale si cerca di duplicare una macchina fisica → se quindi abbiamo un sistema si cerca di creare n copie di questo in modo da poter avere in esecuzione sullo stesso sistema più sistemi operativi
- si crea l'illusione di più macchine che lavorano ognuna su un proprio processore e con la memoria virtuale



→ si ha quindi un determinato strato che permette di avere in esecuzione sulla stessa macchina più sistemi operativi e quindi più kernel!

→ nell'immagine a sinistra è presente una semplice macchina, mentre dall'altra parte abbiamo una macchina virtuale!

→ ognuna della macchine ha un proprio sistema operativo e ha in esecuzione i suoi processi, quindi questi processi vedono soltanto questo kernel al quale sono state fornite l'uso di determinate risorse → sarà quindi la virtual machine implementation a gestire l'accesso a queste risorse hardware → cpu - memoria e gestisce anche la virtualizzazione di vari dispositivi...

→ si partiziona l'uso della cpu tra le varie virtual machine, lo stesso per la memoria → compito della virtual-machine-implementation sarà quello di dare più memoria possibile a tutte ! (tecniche di gestione della memoria)

+il dual-boot non è una macchina virtuale(infatti si usa o uno o l'altro)... se invece è che nello stesso momento puoi usare sia windows che mac allora quello sì!

+ Il sistema operativo crea l'illusione di disporre di più macchine ognuna in esecuzione sul suo processore e con la propria memoria (virtuale)

Le risorse del computer fisico sono condivise per creare le macchine virtuali

→ Lo scheduling della CPU crea l'illusione che ogni VM abbia il proprio processore (o i propri processori)

→ I file system può fornire uno spazio di memorizzazione per ogni macchina virtuale

Parlando un po di storia... sono nate per mainframe IBM nel 1972 (IBM VM/370), oggi ritornate in uso comune (VMware)

→ ad oggi sono utilizzate nei datacenter per utilizzare al meglio le risorse di calcolo

+L'altra caratteristica delle macchine virtuali è che forniscono una completa protezione delle risorse del sistema! → cioè ogni macchina virtuale è isolata dalle altre macchine virtuali...

quindi (riguardando l'immagine) ciò che un processo vede... non è visto da quello di una diversa VM → lo stesso per i filesystem, oppure ancora memoria ecc...

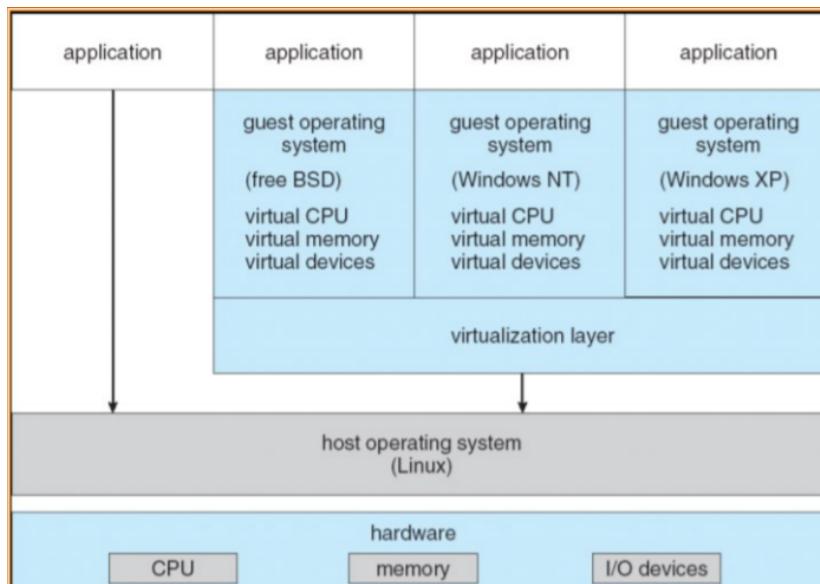
→ ogni ambiente non può interferire con l'ambiente degli altri!

+Macchine virtuale modo perfetto per fare ricerca e sviluppo sui sistemi operativi! → in quanto posso fare sviluppo sulle macchine virtuali invece che sulla macchina fisica in modo da non disturbare il normale uso del sistema!

→ a patto però che creare una VM spesso non è molto semplice → vi è infatti un grosso sforzo per creare un duplicato → ed è difficile farlo anche in modo efficiente!

→ nelle virtual machine quello che si ha è che i kernel sono in esecuzione in modalità utente rispetto al sistema di virtual-machine-implementation → per cui si passa dalla modalità virtual-machine-implementation (la vera modalità kernel) al kernel per i processi e infine i processi che costituiscono la modalità utente

+esempio di virtual machine(virtual box)



→ in realtà si fornisce un sistema di virtualizzazione all'interno delle applicazioni! → dunque risultano in esecuzione in modalità utente nella nostra macchina

→ l'host operating system è quello che viene lanciato di base... dunque abbiamo delle applicazioni che fanno parte del nostro sistema operativo (in questo caso linux)

→ poi abbiamo un sistema che permette la virtualizzazione (ad esempio una virtual box) → che permettono di eseguire delle macchine virtuali su quella macchina che è in esecuzione!

+il problema risulterà quindi quello di come implementare la modalità utente che usa il bit di modo (visto precedentemente)

→ un modo è analizzare il codice precedente che verrà eseguito → codice del sistema operativo in modo tale da cambiarlo/modificarlo e in modo da eseguire le istruzioni che avrebbero bisogno del kernel → le quali vengono tradotte in istruzioni che possono essere eseguite anche in modalità utente!

Container applicativi (saltato nel 2022)

Nei grandi datacenter spesso si devono gestire l'esecuzione di migliaia di dati virtuali, che in realtà erano duplicazioni uno dell'altro...

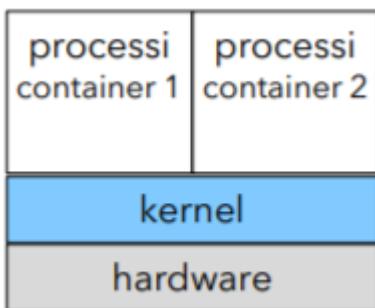
→ Si ha la situazione in cui ogni VM ha un proprio kernel → anche se questi kernel sono spesso identici e per ogni kernel si deve riservare un certo spazio di memoria per appunto gestire un determinato kernel! → questo dunque comporta una limitazione nel numero di VM in base alla quantità di memoria disponibile

→ inoltre far partire ogni macchina virtuale... facendone i boot e varie cose prende comunque tempo...

→ quando dunque si ha necessità di lanciare il tutto in tempi di risposta molto veloci → ovvero macchine virtuali pronte in breve tempo → allora viene usato un approccio a container! → dove non viene creata un'istanza delle nuove macchine ma si sfrutta una sola copia del kernel!

+non si ha un'interfaccia gui ma solo una command line interface → e poi servizi forniti tramite la rete

+Abbiamo dei processi che usano le funzionalità del kernel → il quale sarà singolo!
→ invece di creare la sensazione di avere un nuovo hardware in questo caso viene creata la sensazione di clonare un nuovo sistema operativo → dunque clonare se stesso (e non l'hardware) per una nuova copia del kernel:



→ i processi sono quindi isolati dagli altri container e tra di loro possono comunicare per il modo in cui sono stati progettati → ad esempio sapere che ci sono più processi all'interno dello stesso container ma a priori non sanno dell'esistenza di altri a meno che questi non siamo messi all'interno di una rete!

→ ognuno di questi avrà l'uso della cpu in base a determinate politiche
→ si può fare un partizionamento della memoria per gestire i vari processi all'interno del container → oppure ancora c'è la possibilità di avere delle interfacce di reti virtuali per ogni container → in modo tale che i processi che sono dentro al container sono simili ad una macchina virtuale!

→ il kernel isola i vari processi che fanno parte del container dagli altri processi degli altri container

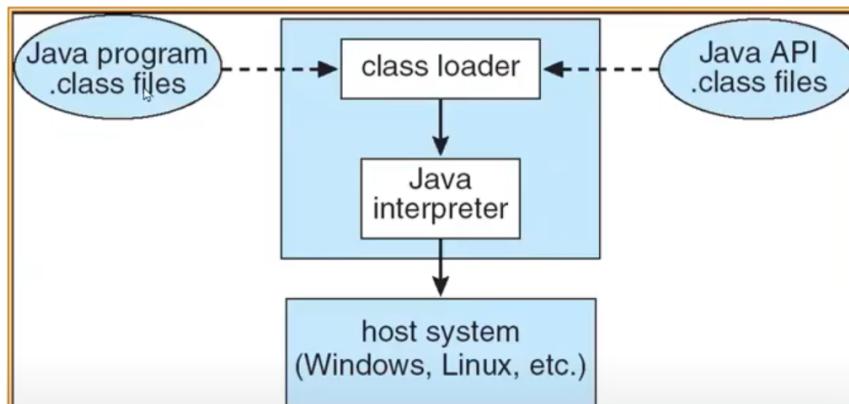
+Vantaggi:

- Minor occupazione di memoria e CPU e overhead di virtualizzazione
- Avvio di un nuovo container molto veloce rispetto ad avvio di una VM → in quanto praticamente risulta come avviare un processo dunque di pochi secondi
- Più facile installare un container → i container hanno un proprio filesystem → e quello che viene fatto è quello di avere filesystem a *layer* nel quale abbiamo una base (che contiene il filesystem di base) e su questa vengono aggiunte delle parti che sono specifiche del filesystem! (ad esempio nuove applicazioni) → in questo modo è più facile far condividere filesystem fra più container (costo minore rispetto a chi li deve duplicare completamente) → e ogni container avrà dei determinati file in più che servono ad un determinato aspetto! (ma il filesystem base rimane lo stesso!)

+Garantire l'isolamento dei vari container non è semplice → in quanto dobbiamo ricordare che sono comunque tutti eseguiti su di un unico kernel! → dobbiamo fare in modo che l'esecuzione di processi di diversi container siano fra loro indipendenti
→ problema che non si pone nel caso delle macchine virtuali dove ognuna ha un proprio kernel! → dove risulta più difficile che i processi comunicano fra di loro

Macchine virtuali JAVA

Ancora un altro modo di virtualizzare → più formalmente è detta *macchina astratta java* → Java infatti permette di eseguire delle applicazioni che sono scritte in un linguaggio macchina che non è il linguaggio macchina di una macchina fisica ma è un linguaggio macchina astratto con una serie di istruzioni codificati con dei filecode → ma che sono comunque eseguibili da una certa macchina virtuale:



→ questa VM (nel mezzo) prende i file punto class che contengono la parte compilata (equivalente dei file object della compilazione del file c), vengono poi caricati insieme alle API del java nella VM e dunque ricompilati ed eseguiti all'interno di questa
→ non è altro che un processo in esecuzione su questa macchina!
→ alla fine l'interprete java usa i servizi del sistema operativo per accedere al filesystem e dunque tutti gli aspetti di sistema (usa ovviamente tutte le chiamate di sistema) per sfruttare tutte le risorse che sono a disposizione

→ la javaVM che è in esecuzione all'interno di un processo ed esegue una particolare applicazione → il vantaggio di questo modo di avere questa macchina virtuale è che può eseguire gli stessi programmi java su macchine host completamente diverse! → ovvero sia su windows, che linux etc... in quanto non vi è bisogno che siano compatibili a livello di object code → in quanto questo sarà ritradotto in linguaggio macchina all'interno della jVM
→ ma non finisce qui... sarà infatti possibile compilare il tutto anche su processori di diverso tipo come ARM oppure x86 → per i quali a seconda dell'utilizzo genereranno dei file di bytecode diversi!

+Macchina astratta perchè non è una macchina fisica! → fisicamente non esegue il bytecode → ma è implementata dentro un processo che legge queste istruzioni (scritte a loro volta in bytecode) e le interpreta o le compila nel linguaggio macchina, della macchina ospite! → la JVM è quindi emulata da una macchina fisica nella quale è in esecuzione

WSL

Windows 10 ha implementato il cosiddetto windows subsystem for linux → che permette l'esecuzione di file linux su windows! → si attiva con una serie di comandi da fare(vedi link su slide) che permette alla fine di usare linux sulla propria macchina

WSL1 (2016) → Implementato come interfaccia, traduce le **chiamate di sistema** linux in chiamate di sistema di windows!

- Problemi con alcune chiamate ad implementare completamente

→ WSL2 (2019) → Usa un vero kernel linux in esecuzione in una VM Hyper-V (il sistema di virtualizzazione di microsoft) → che permette di creare una copia sulla quale è installato un kernel linux in una forma adattata... il componente V di windows gestisce lui la virtualizzazione → per cui assegna una parte della CPU (in base alle politiche presenti) e dunque per eseguire la macchina virtuale ecc...

+docker-desktop → sistema di gestione dei container più usato(in ambito business)

+strace → per vedere chiamate di sistema che un applicazione esegue

+comando top → permette di vedere i programmi/processi che sono in esecuzione in un certo momento → dandone particolari informazioni... ad esempio la quantità di cpu che un processo sta usando(oppure quanto in modalità utente o kernel)

+comando htop → anche questo permette di vedere i processi in esecuzione facendo vedere anche come questi siano in una certa gerarchia dei processi → si vedono inoltre i thread che sono all'interno del processo

→ come prima la percentuale di cpu usata, lo stesso per la memoria → quanto ne è stata allocata, oppure ancora la memoria condivisa ecc...

+è possibile anche da wsl cambiare l'approccio da wsl1 al 2 → nel quale si passa dalla macchina virtuale ad un approccio con le chiamate di sistema tipiche del sistema Linux

→ rifacendo il comando htop si nota come sia la cpu che la memoria siano utilizzate maggiormente in quanto si va a mostrare direttamente lo stato della macchina host!

→ htop risulta in esecuzione come processo di Windows... soltanto che la sua esecuzione

non è gestita dalla macchina virtuale ma è gestita dalla macchina host! (anche se il comando è stato eseguito dalla wsl!)

→ dunque tutte le chiamate di sistema vengono tradotte sulla macchina host e non nella macchina virtuale!

→ nel caso precedente la VM aveva una propria cpu virtuale dunque risultava che non ci fossero processi in esecuzione → le chiamate di sistema fatte nella VM facevano vedere solo quella parte di macchina virtuale!

→ nella wsl2 ho solo uno strato di traduzione dalle chiamate di sistema di linux alle chiamate di sistema linux che mi fanno vedere la stessa macchina! → ambiente più isolato!

Java

Linguaggio di sintassi simile al C ma object oriented pensato come multipiattaforma e **ricco di API standard** (le quali funzionano tutte allo stesso modo nelle varie piattaforme) → funzionalità utilizzabili dalle applicazioni che forniscono molte possibilità, come ad esempio la realizzazione delle applicazioni grafiche...

→ originariamente nato per il web → esecuzioni di applicazioni fornite sul web ed eseguite sui vari client → per questo motivo nacque con determinati vincoli legati alla sicurezza → per strutturare diffusione di virus ecc... (ad oggi non più utilizzata dunque questi aspetti di sicurezza non sono più presenti → anche se sono parti dell'RVM che si occupano di questi)

+Codice java → eseguito sulla Java-Virtual-Machine → macchina astratta con un particolare linguaggio → oltre ad avere il concetto di registri ha anche quello di stack per l'esecuzione delle istruzioni... → i sorgenti (file java) vengono dunque trasformati nel linguaggio macchina di questa JVM → che sono in bytecode → i quali a loro volta sono memorizzati nel file ".class" (equivalente del .o del c) → file che possono poi essere raggruppati a formare una libreria → detti jar (?) → java-archive (che non sono altro che degli zip) → bytecode il quale non viene interpretato ma trasformato in istruzione per il linguaggio macchina (ovvero compilato) → per la macchina ospite!

+Le applicazioni Java sono eseguite da un processo java che esegue il bytecode compilato della applicazione

→ i file .class che contengono il bytecode → possono essere interpretati singolarmente → esiste un programma che prende il bytecode → lo analizza e sa che deve fare

→ per ogni istruzione e quindi esecuzione di questa macchina astratta esegue una serie di istruzioni per interpretarla e poi eseguirla → anche se questa attività risulta dispensiosa!

→ interpretare dei file comporta l'esecuzione di molte istruzioni per analizzare il bytecode per capire cosa deve fare e poi eseguire

→ per cercare di migliorare questa situazione sono nati gli interpreti JVM → i quali in realtà non interpretano ma compilano → prendono il bytecode e lo trasformano direttamente in codice macchina per il sistema ospitante!

→ una volta fatta l'attribuzione del bytecode a linguaggio macchina, questa viene salvata in memoria nella macchina ospite → dunque diventa più performante rispetto alla macchina ospite!

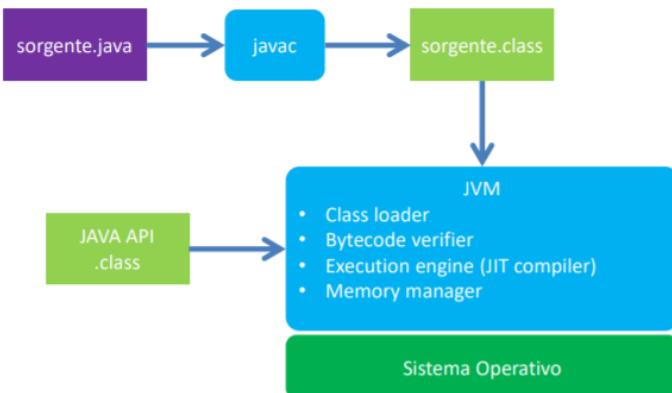
+In java tutto è una classe → non esiste come in C++ la possibilità di avere oggetti, classi, funzioni ecc... il modo per quindi definire il programma principale che deve essere eseguito per far partire un determinato programma è mettere un comando main all'interno di una classe! → ad esempio nella classe HelloWorld mettiamo un metodo statico main → il quale non dipende dalla singola istanza della classe!

→ per stampare qualcosa a schermo (come hello world!) cosa si fa? → dalla classe system si prende l'oggetto out che rappresenta lo standard output su console e poi le dice di stampare (andando a capo) con println → cosa? → Hello world!

+Quando diamo il nome ad un file(o viceversa alla classe) ci deve essere una corrispondenza fra file e nome della classe! → in particolare possono esserci più classi ma solo una classe pubblica la quale deve coincidere con il nome del file!

→ dunque in questo caso si chiamerà HelloWorld.java

+Quale sarà quindi il percorso di un file java?



→ il javac è una sorta di compilatore che prende il file sorgente e lo traduce in uno .class → che contiene il bytecode che esegue una particolare applicazione...
 a quel punto la JVM prende il file .class lo interpreta/esegue prima passandolo al class loader → caricando le classi necessarie all'esecuzione di un certo programma → quindi c'è il bytecode verifier che appunto verifica che il codice esegua quello che deve fare (aspetto legato alla sicurezza delle applicazioni java)
 → a quel punto un altro compilatore (JIT → *just in time*) che compila il codice bytecode in codice macchina e lo esegue!
 → in fine sempre nella JVM c'è un gestore della memoria → il quale si occupa di deallocare quando qualcosa non risulta più necessario! → importante!! → non tutti i linguaggi hanno un gestore della memoria → come C o C++
 +JVM che può usare anche dei file .class caricate dall'API standard che vengono caricate anche loro → per l'esecuzione dell'applicazione java poi alla fine si utilizzerà il sistema operativo per utilizzare le cose necessarie rispetto a cosa deve fare l'applicazione
 → ad esempio nel caso di SO linux, se voglio fare una stampa → si chiamerà una chiamata di sistema *write* per stampare la stringa che si vuole stampare

+Quando si utilizza java in realtà abbiamo a disposizione 2 possibilità → si può solo usare delle applicazioni java, oppure si possono sviluppare delle applicazioni java
 → nel primo caso si utilizza il *Java Runtime Environment* → una cosa che permette l'esecuzione di applicazioni Java → fornisce solo la JVM e dunque utilizzabile solo per eseguire una certa applicazione
 → nel secondo invece avremmo bisogno anche del compilatore → dunque bisogna scaricare il java development kit che contiene tutti i programmi che servono a compilare/dissamblare... in più con il kit viene dato anche un java runtime environment (anche un set di pentole in omaggio) → che dunque permette l'esecuzione dell'applicazione (detta 33232 volte)
 → prodotto il file .class, chiamo il nome della classe dove mi aspetto sia presente il programma main → e lo eseguo!

JVM non solo per java ma anche per altri linguaggi → tipo Scala oppure Kotlin → usato per applicazioni android!

+Esempio programma:

```

public static void main(String[] args) {
    int a=10;
    int b=(a*6)+2;
    System.out.println(b);
}

• istruzioni per macchina astratta che lavora con uno stack, operandi prelevati dallo stack e risultato inserito su stack
• accedono anche a un insieme di variabili locali (indicate con un numero)

```

```

public static void main(java.lang.String[]);
Code:
0: bipush    10
2: istore_1
3: iload_1
4: bipush    6
6: imul
7: istrue_2
8: iadd
9: istrue_2
10: getstatic java/lang/System.out:Ljava/io/PrintStream;
13: iload_2
14: invokevirtual java/io/PrintStream.println
17: return

```

- java compiler prende questo codice qui e lo trasforma nel bytecode!
- i passi sono → metti il valore 10 nello stack, estrai il valore e mettilo in un registro/serie di posizioni dove tenere le variabili locale (e sono indicizzate per posizione) → l'indice mi serve quindi per dare loro una posizione → dunque prendo la variabile a, la metto alla posizione 1 e ne inserisco il valore 10 → a quel punto inserisco il valore della posizione 2 → ovvero a*6 moltiplicato per 2 → serie di passi 4-9 → notare che il comando imul non ha operandi... infatto estrae 2 operandi dallo stack e reinseririsce il risultato nello stack
- a quel punto si mette il valore costante 2 all'interno dello stack! → allo stesso modo iadd lavora come imul soltanto che realizza una somma!
- infine si registra il contenuto che è riposto incima allo stack e viene messo nella variabile locale identificata nel numero 2
- a quel punto si farà l'operazione di stampa prendendo l'oggetto (con getStatic) → accedendo a system.out:Ljava... si carica nello stack il valore nella posizione 2 e si invoca una chiamata all'oggetto con parametro la variabile posta nello stack → eseguendo il println...

17/03/2021

sempre java...

Quali sono i tipi base del linguaggio??

- byte (8 bit, -128 ÷ 127) → valori con segno! (in complemento a 2)
- short (16 bit, -32728 ÷ 32727) • int (32 bit, -2 31 ÷ 2 31 -1) • long (64 bit, -2 63 ÷ 2 63 -1)
- float (32 bit, floating point IEEE 754) • double (64 bit, floating point IEEE 754) → entrambi dunque usano lo standard del floating point → bit di segno - mantissa - esponente
- char (16 bit, codifica UNICODE) → codifica differente da quella del c! → c'è una maggiore varietà per i caratteri che non si limita a quelli della tabella ASCII!!
- boolean (true - false) → occhio che in java non si possono usare gli interi per le clausole if!
- String → è un eccezione rispetto ai tipi base, in quanto le stringhe sono in realtà degli oggetti!

+Come si dichiarano le variabili? → stessa sintassi del C/C++

<tipo> <variabile> [= <espressione>];

→ alcuni esempi sono:

```
int a = 32; int f = 0xff00; /*esadecimale*/ int b = 0b11110000; // binario  
float x = 0.1f; double y = 2.34d; (di default è float)  
char c = 'x'; char nl = '\n'; char alpha = '\u03b1'; //codice unicode → per caratteri speciali non  
presenti nella tastiera
```

String name = "John Xina";

→ l'inizializzazione anche se non esplicita, viene comunque fatta di default per problemi di
“sicurezza” → altrimenti java segnala variabili non inizializzate → per cui in sostanza ogni
variabile\array sarà inizializzata a zero - null - false

→ per le stringhe se non inizializzate esplicitamente, queste vengono poste a null →
riferimento ad un oggetto! → stringa che non punta a niente e non una stringa vuota!

+Come convertire le stringhe in un numero? Alcuni esempi:

- int x = Integer.parseInt("123");
- double d = Double.parseDouble("3.14");
- float f = Float.parseFloat("12.1");

→ tutte bellissime funzioni che uno non userà mai...

+Poi ci sono i soliti operatori presenti dal C

- Aritmetici: +, -, /, *, %
- Bit-wise: &, |, ^, ~ → notare il “ ^ ” fà lo xor! (gli altri sono and, or, not)
- Shift ops: <<, >>, >>> → i primi 2 shift preservano il segno → l'ultimo invece aggiunge
semplicemente un 0 a sinistra per cui si usa nel caso di numeri senza segno
- Logici: &&, ||, ! (operano solo sui booleani!!)
- Confronto: <, <=, >=, >, ==
- Parentesi: (,)
- If in linea: ... ? ... : ... (operatore ternario)
- Assegnamento: =, +=, -=, *=, etc.
- Esempi: • x = (3*y) - 12; • b = ((f & 0xff00) >> 8) | ((g & 0xff) << 8); → per quest'ultimo si fà
inizialmente l'and con la maschera in esadecimale → si shifta il risultato a destra di 8 volte ,
allo stesso modo quell'altro (ma a sinistra) e dunque viene fatto l'or bit a bit → in sostanza si
prende il byte più significativo di f, quello meno significativo di g e vengono poi messi
insieme tramite l'or

+Poi ancora le solite istruzioni del C

- **if(<condizione>) <blocco> [else <blocco>]**
- **while(<condizione>) <blocco>**
- **do <blocco> while(<condizione>);**
- **for(<init>;<condizione>;<incr>) <blocco>**
- **switch(<espr>){**
 case <const>: <istruzioni>
 ...
 default: <istruzioni>
 }
- { <istruzione>; ... }

→ stando però attenti al fatto che le condizioni possono essere fatte solo con dei
booleani!(ovvero non si possono usare condizioni booleane implicite! → tramite solo il valore
degli interi → per usare interi come booleani si usano operatori di confronto)

→ infine le istruzioni vanno tutte messe in delle graffe e separate da dei punti e virgola
+Si usa anche qui la *lazy evaluation* → in una successione di operatori booleani se il primo è falso non guardo il resto!

+stesso discorso ancora per gli array

- `<tipo>[] <variabile> [= new <tipo>[<espr>]];`
- `<tipo>[] <variabile> [= { <espr1>, <espr2>, ... <esprn> }]`

→ alcuni esempi quindi sono:

- `int[] x = new int[100];`
- `float[] ff = new float[n];` → con dentro la dimensione
- `char[] s = { 'a', 'b', 'c', 'd' };`
- `int[][] m = new int[3][2]; //matrice 3x2`
- `float[][] v = { {1.1, 2.2, 3.3}, {4.4, 5.5, 6.6} };` //matrice 2x3 (array di array di float)
- non si possono mettere gli array nello stack (come nel C), ma solo nello heap → tramite la parola chiave new → oppure preinizializzando l'array come (come nel caso di array di char)
- come detto prima i valori non inizializzati sono messi di default a zero (oppure '\0' per i char)

+Come al solito si usano “[...]" per accedere agli elementi con indice da 0 a lunghezza -1; ad esempio: `x[1] = x[0]*2;`

→ inoltre potremmo ricavare direttamente la lunghezza del vettore (non possibile in C) →

– `int l = x.length;` //lunghezza del vettore

→ Se si accede con indice non valido genera una eccezione! → il programma si ferma subito!

+ Gli array sono degli oggetti allocati nello heap, mentre variabili con tipo di base sono nello stack (dunque insieme alle variabili locali) → tutto ciò che è allocato con new va a finire nello heap → dunque tutti gli oggetti (comprese le stringhe!)

+ Attenzione alla assegnazione, copia il **riferimento** non tutto l'array ad esempio:

`int[] x = { 5, 3, 1};`

`int[] y = x;` //copia riferimento → y punta alla stessa zona di memoria di x e non ne crea una nuova!

`y[1] = 2;` → modifco l'array riferito da entrambi!

`System.out.println(x[1]);` //**Stampa 2 non 3!**

+esempio:

- Sommare i numeri passati da riga di comando

```
$java Somma 12 34 450
```

```
Somma.java
public class Somma {
    public static void main(String[] args) {
        int s = 0;
        for(int i=0; i<args.length; i++) {
            System.out.println(args[i]);
            s += Integer.parseInt(args[i]);
        }
        System.out.println("somma=" + s);
    }
}
```

→ notare la funzione parseInt per convertire stringhe in interi (se non riesce a convertire il contenuto in un intero ad esempio se gli passo una lettera → lancia un eccezione!) e per la stampa uso il + in modo da concatenare i due risultati! (infatti il numero contenuto in s è trasformato in una stringa)

+ricordo che per compilare un file java vā eseguito il comando javac nomefile.java → ottendo un file nomefile.class che contiene la codifica bytecode del programma scritto → a quel punto per invocare il programma basterà scrivere java nomefile
 → se quindi il mio programma è una particolare funzione (come nel esempio solo il main) → insieme a java nomefile potro aggiungere dei parametri per il funzionamento del programma!
 → il java è così attento che se passi argomenti sbagliati lancia da sè un eccezione! ad esempio se con il programma somma scrivessi java Somma 10a → eccezione! → non si riesce a interpretare la stringa come un numero!

```
C:\Users\pierf\java>java Somma a1
Programma di somma
a1
Exception in thread "main" java.lang.NumberFormatException: For input string: "a1"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at Somma.main(Somma.java:8)
```

→ la cosa bella è che fà anche vedere dove è stata lanciata l'eccezione ripercorrendo lo stack!
 → occhio al fatto che il nome della classe deve corrispondere al nome del file! → regola importante

+Qualcosa in più sulle stringhe

- Sequenze di caratteri **immutabili** → dunque non modificabili!
 - String a = "Hello "; – String name = "John"; (risultano costanti)
 - String msg = a + name + "!"; //concatenazione di stringhe! → alla fine msg riferisce la stringa hello john!
 - int l = msg.length(); //lunghezza della stringa
- +è possibile concatenare stringhe con tipi base come interi!
 - Anche le stringhe sono oggetti memorizzati nello heap! (come del resto qualsiasi altro oggetto!) → non esiste avere oggetti il cui contenuto è memorizzato nello stack!

- Non sono array! (array infatti non sono oggetti ma simili)
- +Modificare una stringa significa in realtà creare una nuova stringa a partire dalla prima...

→ ATTENZIONE al confronto! → non si confronta il contenuto di una stringa ma il loro riferimento!!

```
String a = "1234"; String x = "12";
```

String b = x + "34"; → b contiene "1234"

```
if(a == b) //confronta i due puntatori (e non il contenuto delle stringhe!)
```

```
    System.out.println("Uguali!");
```

→ Il confronto a == b difficilmente sarà vero. (solo nel caso si riferiscano alla stessa zona di memoria)

→ nel caso ad esempio scrivessi String a = "1234"; String x = "1234"; in quel caso java molto furbescamente non va a riferirsi ad una nuova zona di memoria ma riusa la stessa!

(l'oggetto è creato unicamente una volta sola) → in quel caso a == x da vero!

→ dunque per confrontare il contenuto di 2 stringhe Si usa: if(a.equals(b)) ...

```
if(a.compareTo(b) < 0) //si usa per comparare due stringhe → dà zero se stringhe sono uguali, -1 nel caso di a < b o 1 se a > b!
```

+un altro esempio:

- Cercare una stringa in un array di stringhe.

```
String[] mesi = {"gen", "feb", "mar", ...};  
String m = "mar";  
int mese = 0;  
for(int i=0; i<mesi.length; i++) {  
    if(mesi[i].equals(m)){  
        mese = i+1;  
        break;  
    }  
}  
if(mese==0)  
    System.out.println("mese non valido");  
else  
    System.out.println("mese: "+mese);
```

→ quando fà la concatenazione (come già detto) porta mese da intero a stringa e poi fà la concatenazione voluta

→ notare che se avessi provato a fare "mesi[i] == m" potrei aver ottenuto vero! → in quanto nell'array mesi definisco ad esempio 'mar', dunque quando creo il nuovo oggetto m questo punta di già al riferimento creato prima! → dunque le 2 locazioni coincidono!

→ non conviene comunque usare "==" per le stringhe che potrebbe portare ad errori non subito riconoscibili!

Garbage collector

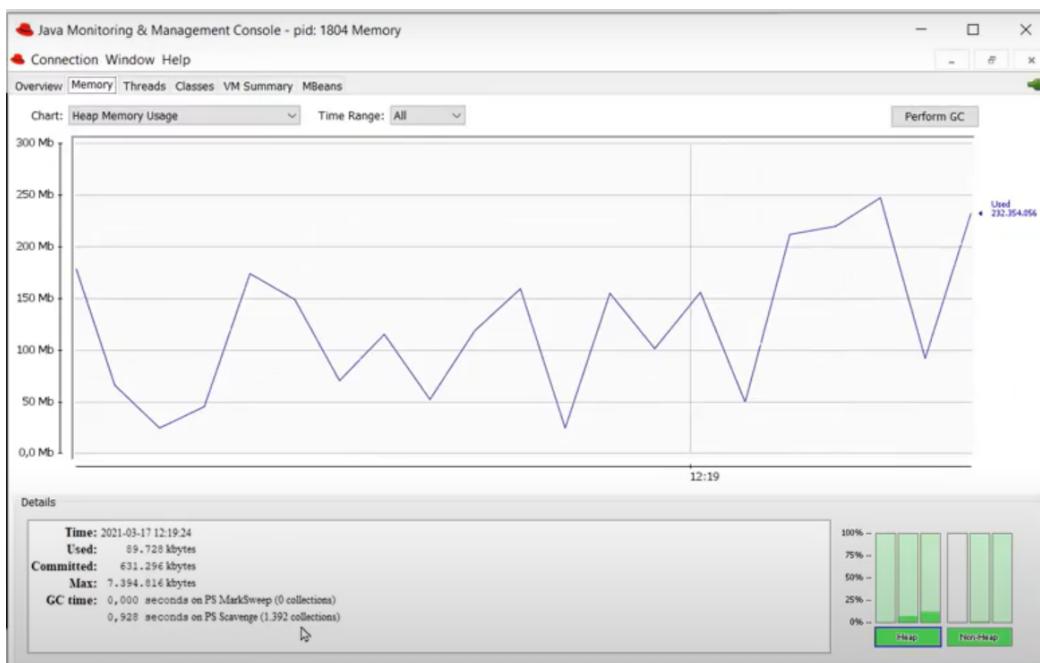
- Gli oggetti creati con operatore new non devono essere distrutti esplicitamente (come in C++) → non esiste infatti un operatore delete!

il Garbage Collector si occupa di liberare lo spazio di memoria non più raggiungibile.

→ gestisce quindi i riferimenti agli oggetti dello heap ed è quindi suo compito scoprire se delle zone degli oggetti allocati non sono più riferiti → ovvero non sono più utilizzabili e quindi vengono rilasciati

→ E' una attività eseguita quando c'è bisogno di memoria. → quindi viene eseguita "ogni tanto" o comunque quando si neccessita di memoria → entra in esecuzione andando alla ricerca di zone di memoria non più riferite e vengono liberate

→ Si può usare il programma jconsole (fornito con il JDK) per vedere l'occupazione di memoria della JVM (a seconda dell'esecuzione di un certo processo)



→ GC time dice quanti secondi impiega per fare la pulizione e quando collection ha fatto
→ intervallo tipico del dente di sega? → arriva ad un certo valore di picco e poi decide di attivare la GC → fare del GC porta via del tempo per questo motivo non è fatto continuamente ma ogni tanto! (e quando uso della memoria troppo elevato)

+Come vengono eseguiti gli algoritmi di garbage collector varia a seconda della JVM... si ha comunque l'obiettivo di ridurre il più possibile il tempo di esecuzione del garbage collector → in modo da renderlo quasi trasparente!

+Nel caso avessimo un applicazione continua → ad esempio un server che non viene mai fermata → in quel caso lì se abbiamo un memory leak crea poi dei problemi!! → memorizzando sempre più roba senza mai cancellarla porta ad un crash dell'applicazione!!
→ questa risulta tempo-variante e soprattutto trovare dei memory leak non è semplice!
→ per questi motivi linguaggi come java sono spesso usati per la gestione di applicazioni server... i quali garantiscono la gestione della memoria in modo autonomo e garantiscono un certo livello di affidabilità/permanenza

Classi e oggetti

→ anche per la parte di object oriented, java riusa gli stessi concetti del C++, se non per alcuni aspetti differenti...

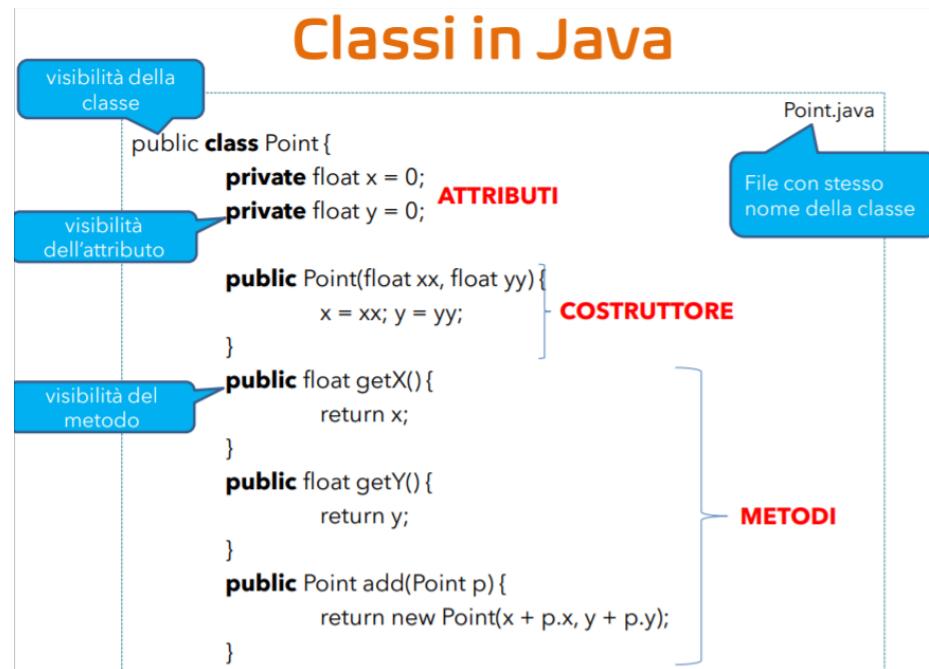
Una **classe** rappresenta un insieme di entità che condividono delle stesse caratteristiche → rappresentate tramite degli attributi (singole proprietà che la classe ha)

→ in più presenti delle funzionalità che possono essere eseguite sulle istanze di un determinata classe → ad esempio: automobile

- Gli oggetti sono gli elementi che fanno parte di questi insiemi e sono detti istanze della classe

+ Come si definisce una classe in java?

→ attenzione → si fa tutto in una singola classe! (più file si metteranno solo all'interno di un package)



→ della classe `Point` ci saranno quindi determinate istanze che rappresentano un particolare punto → queste hanno quindi particolari valori degli attributi e possiamo usare una serie di funzionalità che possiamo usare con questa classe

→ esiste anche qui il costruttore → metodo particolare per inizializzare gli attributi della classe

→ e metodi getter/setter per accedere/manipolare lo stato dell'oggetto

→ notare anche la parola chiave `class` prima di `Point` per esplicitare la visibilità della classe → `public` visibile a tutti, se non è `public` è visibile solo a livello di `package`

→ possiamo anche specificare la visibilità per gli attributi e i metodi → la visibilità privata → implica che sono visibili solo esclusivamente all'interno della classe → ovvero i metodi di questa classe (vedo bene sta cosa con il costruttore)

→ la visibilità va specificata per ogni elemento all'interno della classe!

+ L'operazione di somma in realtà va a creare un nuovo punto → una volta creato un nuovo punto infatti non c'è la possibilità di modificare il suo contenuto! → in questo caso infatti non ho definito dei metodi che ne modificano lo stato del punto → ma ho solo metodi per accedere o creare nuovi punti!

+ Notare che diversamente da C++ la visibilità dev'essere ripetuta per ogni elemento!

+Definita quindi la classe point decido di utilizzarla in una successiva
PointsProgram.java

```
class PointsProgram {  
    public static void main(String[] args) {  
        Point p = new Point(1,2);  
        p.x = 3; //ERRORE, attributo x è privato  
        p = p.add(new Point(1,1));  
  
        System.out.println(p.getX() + "," + p.getY());  
        p = null;  
    }  
}
```

→ creato il nuovo punto con new, passo alla variabile p il suo riferimento!

→ come detto in precedenza il metodo add creerà un nuovo punto di coordinate 2,3 → dunque il riferimento a questo oggetto va a p → la reference precedente viene persa e dunque ci penserà il garbage collector successivamente a eliminarla! (sia per il punto p iniziale che il punto (1,1))

→ Accedo con i metodi getter al contenuto x e y del punto per stamparli e decido di terminare il programma dando un riferimento nullo alla variabile p → in modo che non punti più a niente! → in modo simile ad un puntatore! (concettualmente)

→ La variabile p è una reference ad oggetto di tipo Point → simile quindi alle reference del C++ soltanto con la caratteristica aggiuntiva che può puntare a null!

→ Gli oggetti associati alle reference sono allocati esclusivamente nello Heap

→ se però una reference è null e si chiama un metodo usando la reference viene generata una eccezione «null pointer exception» che blocca il programma → dunque vā in qualche maniera gestita! → ad esempio fare p.getX(), dopo aver posto p = null genera un eccezione!

+Come nel caso delle stringhe occhio ai confronti... se voglio valutare se due oggetti hanno lo stesso contenuto non bisogna usare “==” → che mi andrebbe a confrontare le 2 reference!

→ dunque un confronto del genere risulta vero solo se i due oggetti hanno stesso riferimento in memoria!

```
Point p1 = new Point(1,1);
```

```
Point p2 = new Point(1,1);
```

...

```
if(p1==p2) ... //questo è falso, p1 e p2 puntano a due oggetti diversi
```

(se invece faccio point p1 = new point(1,1) e point p2 = p1 allora il confronto sarà vero!)

+Occhio però che in java non è possibile ridefinire gli operatori come nel C++

Passaggio dei parametri

I parametri di un metodo sono passati:

– **per valore** i tipi di base → interi -float - char - esclusi stringhe e array (e gli oggetti in generale) vengono passati per valore → sono messi nello stack, vengono quindi presi e messo il valore direttamente → per ragioni di efficienza

– per riferimento oggetti, stringhe ed array... preso ad esempio:

```
void calcola(int a, float[] b, String c) { a = a * 2; b[0] = 1; c = c + "d"; }
... int v = 10; float[] x = new float[5]; String s = "abc"; calcola(v, x, s);
```

→ essendo a una copia del valore passata nel programma chiamante, non prova nessuna modifica alle variabili usate per il passaggio → quindi nell'esempio "a" rimane 20 solo nella funzione!

→ mentre per l'array b passato per riferimento → nell'esempio b e x si riferiscono alla stessa zona di memoria per cui il cambiamento fatto nel programma chiamante rimane fissato! → non si è copiato l'array ma gli ha passato il riferimento!

→ caso particolare sono infine le stringhe → in questo caso c risulta essere una variabile locale che si riferisce ad una stringa → viene quindi modificato il riferimento che c'è in c (concatenato con la stringa "d") → dunque quando passo il valore della stringa s nel metodo calcola → poiché le stringhe abbiamo detto sono immutabili... creo una nuova stringa! → c si riferisce ad una nuova stringa che ha come contenuto "abcd" , mentre la variabile s continua però a riferirsi ad una stringa che ha contenuto "abc" ! (anche dopo l'esecuzione del metodo calcola ... viene impedita la modifica)

→ le stringa c è dunque come se fosse stata passata per valore, ma solo per il fatto che c non risulta modificabile! (dunque per stringhe sempre passaggio per riferimento ma opera come passaggio per valore!)

→ per modificare quindi il contenuto della stringa o posso aggiungere un return nome della stringa, oppure ancora peggio decido di passarlo ad un array di stringhe dove però è presente un solo elemento → ovvero la stringa che vuoi modificare!

Visibilità

- private: visibile solo dai metodi della classe
- protected: visibile dalla classe, dalle classi derivate e dalle classi dello stesso package → le classi in java si possono infatti raggruppare in dei pacchetti e le classi che appartengono allo stesso pacchetto danno un livello di accessibilità maggiore in quanto quelle classi lì in realtà sono state progettate per collaborare ! → dunque hanno più possibilità di accedere a cose private/nascoste
- public: visibile da tutti
→ se omesso è visibile dalle classi all'interno dello stesso package! → Una classe può essere solo public o visibile all'interno del package (senza indicare la visibilità)

+ Il nome del file java deve corrispondere al nome dell'unica classe pubblica presente al suo interno → ad esempio nel file memory.java ci può essere solo la classe pubblica memory, le altre classi saranno invece accessibili solo a livello di package! → notare che non c'è bisogno di include o altre cose per classi definite allo stesso livello → sono tutte visibili all'interno del package

+Esempio:

```

public class IntList {
    private int value;
    private IntList next = null;
    public IntList(int value, IntList next) {
        this.value = value; this.next = next;
    }
    public IntList(int value) {
        this(value, null);
    }
    public IntList add(int v) {
        if(next == null) next = new IntList(v);
        else next.add(v);
        return this;
    }
    public String toString() {
        if(next==null)
            return "" + value;
        return value + ", " + next.toString();
    }
}

```

→ al posto dei puntatori per le liste si usano i riferimenti!!

→ notare l'overloading del costruttore → il secondo infatti richiama il primo costruttore → attraverso il costrutto this passando un valore e null! → in java infatti non esistono i parametri di default!

→ il metodo add invece controlla che il prossimo elemento della lista sia null → a quel punto sono arrivato alla fine della lista e inserisco il nuovo oggetto → mettendo nel next il riferimento ad una nuova lista che contiene il valore v (notare che per la creazione del nuovo oggetto si utilizzi il secondo costruttore!), altrimenti ricorsivamente si passa all'elemento successivo fino ad arrivare all'ultimo elemento e quindi aggiungere il nuovo elemento → il metodo quindi ritorna una intList e per farlo si ritorna this → questo permette di fare il *method chaining* che permette di chiamare su un oggetto più metodi in cascata in modo più comodo (semplificazione sintattica)

→ anche il metodo *toString* agisce ricorsivamente... trasformando in stringa tutti gli elementi successivi fino a che next = null → i valori della stringa saranno quindi presentati separati da delle virgolette! → notare l'uso di return ""+value → in modo per fare la trasformazione da intero a stringa e dunque lo posso ritornare!

ecco quindi un esempio d'uso della lista

```

IntList l = new IntList(3).add(2).add(1);
System.out.println(l);
l = new IntList(4, l);
System.out.println(l);

```

→ inizialmente creo la lista di valore 3, aggiungo poi 2 e richiamo nuovamente la stessa lista mettendoci 1 → per il method chaining l'add ritorna se stesso, dunque ritorna l'oggetto che è stato appena creato!

→ vedo adesso la semplificazione detta prima → ovvero invece di separare le chiamate ai metodi → ovvero IntList l = **new** IntList(3) poi l.add(2) e infine l.add(1) → faccio tutto su di un'unica riga!

→ quando quindi eseguo la stampa vado direttamente a chiamare il metodo *toString()* che in realtà è un metodo particolare di java → che è ereditato dalla classe object → e chiamato

quando passato un oggetto a `println!` → dunque non importa scrivere `l.toString()` → ma lo chiama da sè!

→ la chiamata finale del print mi darà “ 4,3,2,1” (faccio l'inserimento in testa!)

Overloading

Posso definire metodi con lo stesso nome ma con parametri in numero e di tipo diverso ad esempio: – Point add(Point p) ... – Point add(float x, float y) ... dunque invece di scrivere
– `p = p.add(new Point(1,2));` scrivo `p = p.add(1,2)`

+Inoltre abbiamo già visto la possibilità di fare overloading del costruttore, spesso usati dovuti al fatto che non possiamo usare i parametri di default! → dunque definisco le varie varianti con nuovi argomenti...

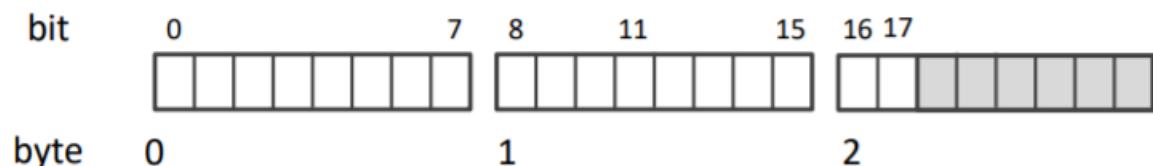
+Alcuni IDE per java sono... – eclipse (sconsigliato da pierfrancesco) – intelliJ IDEA – Apache NetBeans (consigliatissimo da pierfrancesco che dice si trova bene)

+Un primo esercizio...

Realizzare un array che contenga n bit, in modo che si possa impostare ed accedere ad ogni singolo bit e che usi la minore quantità di memoria possibile.

→ si usano gli operatori bit-wise e shift operators per modificare il singolo bit di un byte!
→ alla fine inoltre testare il programma definendo un array di 1001 bit e impostare a 1 i bit in posizione pari e 0 i bit dispari

→ se prendo ad esempio N = 18 bit → approssimando per eccesso, mi serviranno 3 byte!



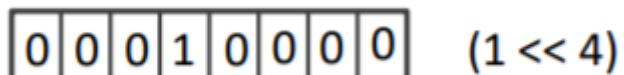
→ quelli in grigio non saranno utilizzati(fra questi dovrebbe esserci anche la casella 17) ... per riferirmi quindi a qualsiasi posizione del byte basterà fare $p/8$ che trova il byte dove si trova il bit da modificare → ad esempio quelli da 0-7 danno 0, da 8 a 15 → 1 e così via...
→ mentre per trovare la posizione del bit esatta, trovato il byte, mi basterà partire da destra e scalare di quante posizioni mi serve → ad esempio 11 → da 15 conto 0,1,2,3,4 e trovo l'11!
→ faccio questa operazione con il modulo!

Es: $p=11$

$$11/8 = 1$$

$$7 - 11 \% 8 = 7 - 3 = 4$$

→ questo quattro sarà l'x successivo



Preso quindi un nuovo array del tipo

→ dove ho messo 1 e l'ho shifatato a destra di 4 posizioni (in realtà è un byte) → dunque farò l'and fra questo byte e quello in cui voglio conoscere il contenuto in posizione 11

- se ottengo un numero positivo → nella posizione 11 c'era un uno, altrimenti uno zero
- dunque per leggere lo stato nella posizione x (al byte b) farò “ $b \& (1 << x)$ ” con end commerciale operatore bit a bit
- per impostare invece a 1 un bit in posizione x in byte b farò “ $b = b | (1 << x)$ ” (con l'or gli altri valore rimangono gli stessi! → uso sempre il byte di prima)
- allo stesso modo per impostare a 0 farò l'operazione di and con stavolta il negato del byte creato prima! → “ $b = b \& \sim (1 << x)$ ” → il negato del byte di prima sarà infatti

1	1	1	0	1	1	1	1
---	---	---	---	---	---	---	---

$\sim (1 << 4)$

- per cui lascio invariati tutto il resto e cambio solo dove ho lo zero!

- la soluzione finale risulta quindi

```

public class BitArray {
    private byte[] bits;
    public BitArray(int nbits) {
        bits = new byte[(nbits+7)/8];
    }
    public boolean get(int bit){
        return (bits[bit/8] & (1 << (7-bit%8))) >0;
    }
    public BitArray set(int bit, boolean v){
        if(v)
            bits[bit/8] |= 1<<(7-bit%8);
        else
            bits[bit/8] &= ~(1<<(7-bit%8));
        return this;
    }
}

```

- con il costruttore quindi uso l'operazione di array di byte → in modo che abbia l'arrotondamento per eccesso!
- quindi abbiamo i metodi per l'accesso(se maggiore di zero è uno altrimenti è zero) e settare un bit (se v = true allora uso l'or altrimenti l'and) → secondo le operazioni viste prima

+Veniva quindi chiesto di fare un test del programma...

```

public class BitArrayTest {
    public static void main(String[] args) {
        BitArray bits = new BitArray(1001);
        for(int i =0; i<1001; i++)
            bits.set(i, i%2==0)
        for(int i=0; i<1001;i++)
            System.out.print(bits.get(i)? "1" : "0");
        System.out.println();
    }
}

```

- inizializzo l'array con 1 nel caso sia pari e zero se dispari
- notare l'uso della funzione print → che permette di stampare senza andare a capo! → a seconda di bit a 1 → vero o 0 → falso stampo “1” o “0”

→ notare anche che potrei sostituire il secondo ciclo for che stampa i bit, aggiungendo un metodo `toString()` all'interno della classe `bitArray`, in modo tale che una volta scritto mi basterà nel main fare la chiamata `System.out.println(bits)` → che invocherà proprio il metodo appena definito automaticamente!

(direi che quando sarà il momento di fare sistemi operativi provare i programma non sarebbe male → magari farci sopra pure il debugging!)

Metodi e attributi statici

I metodi o attributi statici sono delle caratteristiche delle classi e quindi associati all'intera classe ma non al singolo oggetto

→ Gli attributi statici sono in sostanza delle variabili globali e i metodi statici sono delle funzioni che possono essere chiamate in qualsiasi punto del nostro programma senza avere uno specifico oggetto sul quale essere chiamati!

Ad esempio:

```
class Global { private static int time = 100; } → attributo statico accessibile solo all'interno di questa classe!  
public static void setTime(int t) { time = t; }  
public static int getTime() { return time; } → questo permette di accedere al time statico  
} ... Global.setTime(Global.getTime()+1); → per accedere a dei metodi statici lo si fa usando l'attributo della classe!
```

→ gli attributi statici sono accessibili anche dai metodi non statici, ma all'interno di un metodo statico posso accedervi soltanto a caratteristiche statiche di una classe! → non posso usare attributi che parte di un'istanza/oggetto in quanto quando chiamo un metodo statico, lo chiamo sulla classe e non sulla singola istanza di oggetto!

→ per accedere a metodi/attributi d'istanza per quelli lì ci dovrei accedere usando l'istanza! (tramite un'istanza posso comunque accedere ad attributi (o forse metodi) statici → anche usando il nome dell'istanza)

+slide su classe math

- La classe `Math` di java definisce una serie di metodi statici che implementano le funzioni matematiche usate comunemente (`sqrt`, `pow`, `sin`, `cos`, `log`, `exp`, `min`, `max`, `random`, ...)
- `double x=Math.sin(2*Math.PI*a/360);`
- `double v = Math.random();`
//numero pseudo casuale in [0,1]

t

Gerarchia delle classi

Una classe può essere derivata da un'altra classe estendendone le caratteristiche (attributi o metodi) → in java però diversamente dal C++, esiste solamente l'ereditarietà singola → quelle multipla esiste in modo più limitato

→ dunque possiamo estendere da una sola classe!

→ la sintassi è

class <nome_classe> [**extends** <nome_classe>] {

...

→ con nome della classe estesa che deve essere unica (si ovviamente a questo problema sfruttando le interfacce)

+Se omesso extends la classe viene derivata dalla classe predefinita Object! → la quale rappresenta qualsiasi oggetto che deve essere gestito da java → avevamo infatti usato dei metodi precedentemente che non erano stati dichiarati ma in realtà presi dalla classe object!
→ ad esempio il `toString()` è un metodo che è definito nella classe object e possiamo ridefinire nelle classi derivate!

+La classe derivata possiede tutte le caratteristiche della classe padre. → A questa sono aggiunte le caratteristiche specifiche (di solito nuove caratteristiche)... ad esempio:

```
class Persona { public String nome, cognome; ... }  
class Studente extends Persona { public String nmatricola; ... }  
→ nmatricola è un nuovo attributo → studente ha sia nome, cognome che nmatricola → un riferimento ad un oggetto della classe studente a questo posso accedere a tutti gli attributi visti (purchè questi siano public)  
→ continuando con l'esempio... Studente s = new Studente();  
s.nome = "Paolo"; s.cognome = "Rossi"; s.nmatricola = "12345678";
```

→ un riferimento ad uno studente lo posso assegnare ad una persona → posso quindi fare Persona p = s; → infatti studente si riferisce ad s, ma studente è anche una persona → quindi p ha tutte le caratteristiche di quella persona (è un riferimento valido)
→ tramite p posso accedere ai due attributi(ma non a matricola) scrivendo ad esempio "System.out.println(p.nome+" "+p.cognome);"

+A caso si ritorna alla visibilità

Attributi e metodi dichiarati con visibilità protected sono accessibili dalle classi derivate e dalle classi dello stesso package. → dunque se avessi dichiarato nome e cognome protetti questi sarebbero stati accessibili anche da studente! (ciò non sarebbe lo stesso con visibilità privata → accessibili solo da personali!)

Eredità e costruttore

Riprendendo lo stesso esempio di persona - studente, e aver definito nella classe base il costruttore

```

class Studente extends Persona {
    private String nmatricola;
    public Studente(String nm, String cg, String mt) {
        super(nm,cg);
        nmatricola = mt;
    }
    ...
}

```

Chiama il costruttore della classe Persona

tramite quindi la keyword super mi riferisco alla superclasse della classe studente che è appunto persona → avrei comunque potuto scrivere nome = nm e cognome = cg → in quanto protette (nell'esempio sopra sono protette), ma se le avessi dichiarato in persona come privati → l'unico modo per inizializzare questi sarebbe stato il costruttore della classe base!

Polimorfismo

I metodi in Java sono tutti principalmente polimorfici dunque c'è la possibilità di fare il cosiddetto override del metodo (della classe base) → facendo l'override dunque dà un ulteriore specificazione del metodo ad esempio:

```

class Persona {
    ...
    public void print() {
        System.out.println(nome+" "+cognome)
    }
}

class Studente extends Persona {
    ...
    public void print() {
        super.print();
        System.out.println(nmatricola);
    }
}

```

Chiama metodo print della classe padre

→ notare che il metodo print() ha sia lo stesso nome, che anche la stessa visibilità
+Quando quindi eseguo Persona p = new Studente("mario", "rossi", "12345");
p.print(); //chiama Studente.print() e non Persona.print() → in quanto mi **riferisco** ad uno studente come se fosse una persona e siccome ho ridefinito un comportamento di p → in realtà vado ad attivare quello più specifico!

Quando si fa override di un metodo in una classe derivata si può aggiungere l'annotazione opzionale @Override → Rendendo esplicita l'intenzione di ridefinire un metodo!

→ A questo punto il compilatore controlla se esiste il metodo nella classe padre ed è compatibile (stessi tipi di parametri) e in caso non ci sia fallisce (senza @Override, ce ne saremmo accorti solo all'esecuzione)

Classi Astratte

Sono classi che non possono avere istanze dirette ma si possono solo **derivare** altre classi!
→ si definisce una classe astratta semplicemente con la keyword **abstract** → ad esempio

public abstract class Shape { ... } → ci saranno infatti varie forme geometriche che saranno istanze dello shape → il quale rappresenta quindi il concetto generale → che può avere rappresentazioni specifiche come cerchio, quadrato ecc..

→ viene quindi definito anche un metodo astratto che mi servirà specializzarlo successivamente → ad esempio public abstract void draw(); → il metodo dovrà essere implementato nelle varie classi derivate → ad esempio:

```
public class Rectangle extends Shape {
    private int width;
    private int height;
    public Rectangle(int x, int y, int w, int h) {
        this.x=x; this.y=y;
        this.width = w; this.height = h;
    }
    public void draw() {
        ...
    }
}
```

→ con x e y attributi della classe base → se non avessi ridefinito il metodo draw(), il compiler genera errore!

→ stesso esempio poi posso fare anche con il cerchio (notare che l'estensione è sempre di tipo pubblica!)

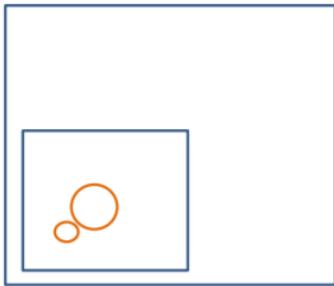
+Another example:

```
public class CompositeShape extends Shape {
    private Shape[] shapes;
    private int n;
    public CompositeShape(int nshape) {
        this.n = 0;
        this.shapes = new Shape[nshape];
    }
    public CompositeShape add(Shape s) {
        this.shapes[n++]=s;
        return this;
    }
    public void draw() {
        for(int i=0; i<n; i++)
            this.shapes[i].draw();
    }
}
```

→ compositeShape composta da un insieme di shape → sfrutto la generalità di shape per avere un array di qualsiasi tipo di shape! → potrò dunque metterci qualsiasi cosa che posso derivare da shape!

→ aggiungo quindi l'attirbuto n che mi dice il numero di shape, nel costruttore lo inizializzo e vado anche ad allocare un nuovo array → notare però che in java, questo non è un array di una certa dimensione ma un riferimento ad un array che non è stato inizializzato! → dovrò quindi allocare questo esplicitamente → nel costruttore indico l'array di shapes come riferimenti a shape di dimensione nshape → questo vale per qualsiasi array di oggetti → non ci vanno le variabili come nel caso di array di int-float ecc... ma solo i riferimenti!!

- il metodo add quindi prende una shape astratto e lo aggiunge all'array in fondo, ritornando l'oggetto stesso (anche se non viene fatto nessun controllo!)
 - se infatti metto più valori di quanti ne avvessi dichiarato alla fine mi darà un eccezione! (se vado oltre la dimensione dell'array)
 - nel metodo draw invece uso proprio l'override! → itero sul numero effettivo di elementi e a seconda di avere un cerchio/rettangolo disegnerò appunto un cerchio o un rettangolo!
 - +Esempio di uso:
- ```
CompositeShape s1 = new CompositeShape(10);
s1.add(new Rectangle(10,10, 100,100)); s1.add(new Rectangle(20,20, 50, 50));
CompositeShape s2 = new CompositeShape(3); → al massimo 3 elementi all'interno
s2.add(new Circle(50,50,5)); s2.add(new Circle(60,60,15));
s1.add(s2); → posso fare questo in quanto anche compositeShape è uno shape! → lo posso quindi mettere all'interno di un composite... ho creato una sorta di gerarchia dove composite... può contenere altre shape
s1.draw(); → chiamo il draw di tutti gli elementi al suo interno → disegno quindi i 2 rettangoli e quindi i 2 cerchi all'interno dell'oggetto s2 → che a sua volta un compositi shape!
→ esce fuori questa figurina molto carina:
```



## Interfacce

- Abbiamo detto che Java non permette ereditarietà multipla (per problemi dovuti a gestione attributi nell'ereditarietà a diamante) → si usano quindi le **interfacce** → le quali sono delle classi le quali definiscono solo delle funzionalità! → ovvero non hanno attributi! (al massimo possono essere presenti degli attributi ma questi devono essere statici)
- le interfacce danno la possibilità di definire l'insieme delle funzionalità che devono essere implementate per poter interagire con un insieme di classi
  - i metodi sono implicitamente astratti e pubblici → i quali non hanno implementazione ma dovranno essere appunto ridefiniti!
  - come detto prima possono avere metodi e attributi statici e inoltre possono essere derivate da altre interfacce (non c'è un limite)
  - ma la caratteristica importante è che una classe può essere derivata da più interfacce! → infatti questa non crea nessun problema!
  - la sintassi è simile a quella delle classi e usa la keyword interface

```
[public] interface <Nome> [extends <Nome>[,<Nome>,...<Nome>]] {
 <metodi astratti pubblici>
 <attributi statici>
 <metodi statici>
```

1

- i metodi presenti o sono astratti (e pubblici) oppure statici...

Una volta definiti si possono usare all'interno della definizione delle classi e si scrive

```
[public] class <Nome> [extends <Nome>] [implements
<Nome>[,<Nome>...]] {
 ...
}
```

extends → da una classe, implements una o più interfacce → la classe deriva da una classe base e in più implementa tutti i metodi(tutti astratti) definite in determinate interfacce → tutti i metodi astratti dovranno essere necessariamente ridefiniti in questa classe!

+Esempio:

```
public interface MovableObject {
 void move(int dx, int dy);
}

class Rectangle extends Shape
 implements MovableObject, Cloneable {
 ...
 public void move(int x, int y) {
 ...
 }
 }

MovableObject mo = new Rectangle(...)
mo.move(10,20);
```

Metodo astratto  
e pubblico

→ potrò quindi dire che rettangolo è sia uno shape che un movableObject( che cloneable) → dovrò quindi implementare tutti i metodi dichiarati in movableObject e clonable, prima di usare un oggetto rettangolo → a quel punto posso avere un riferimento a quella interfaccia!  
→ usando il riferimento a *mo* posso usare il metodo move!

## 24/03/2021

### Cast

- L'operatore di cast serve a trasformare una istanza di un tipo in altro tipo analogo ma meno preciso → ad esempio long l = 10; int a = (int) l; → passo da qualcosa a 64 bit a 32! → un assegnazione diretta potrebbe portare a perdita di precisione → per cui il programmatore deve essere esplicito se vuole perdere questa precisione
  - E' necessario un cast nella trasformazione da int a byte, da int a short, da double a float etc.
  - lo stesso operatore di cast lo si utilizza anche per trasformare una reference di un tipo ad un altro! → con però delle limitazioni → si può fare solo nel caso in cui la reference che voglio trasformare sia una reference ad una classe padre e voglio trasformarla in una reference a classe derivata, ad esempio
- Persona p = new Studente(); ... Studente s = (Studente) p → tutto torna poichè p si riferisce inizialmente ad uno studente(o si poteva riferire anche ad una sua classe derivata)... ma se non fosse così viene generata un eccezione!!

→ il cast è necessario quando si vuole usare un metodo della classe derivata → ad esempio se tramite p voglio accedere al numero di matricola non posso farlo, se non facendo il downcast!

→ notare che nell'esempio anche se creo uno studente ma uso una variabile persona, allora di quell'oggetto potrò accedere solo agli attributi/metodi che le due classi hanno a comune!  
→ tramite p accedo solo nome e cognome!

+In Java E' possibile anche controllare se una reference punta a una istanza di una classe (o sua derivata) tramite operatore **instanceof** → è un operatore booleano che quindi mi dice se una reference è un'istanza di una certa classe... ad esempio

Studente s = null; if(p instanceof Studente) s = (Studente) p; → per cui p punta ad un oggetto della classe studente oppure ad una sua derivata! → in questo modo possiamo quindi eseguire un cast in sicurezza non rischiando di generare un'eccezione (utile nel caso di gerarchie molto grandi di classi!)

## Final

Nel java non sono presenti le *costanti*, tramite quindi il modificatore *final* possiamo indicare una variabile, un attributo o anche un metodo non più modificabile → questo risulta quindi finalizzato → modificatore anche usato per dichiarare delle costanti

→ Ad esempio: final int A = 1000;

final float PI; PI = 3.14; //da questo punto in poi PI non è modificabile

→ un attributo final, static e public viene usato in una classe per indicare una costante! → aggiungiamo infatti public e static in modo che sia "usabile", in qualsiasi parte del codice!

+ Un *metodo final* invece indica un metodo che non può essere **ridefinito** in una classe figlia. → se tento di fare l'override mi da errore!

→ allo stesso modo si possono avere anche classi final da qui non è possibile derivare altre classi

## Package

I package servono a raggruppare classi semanticamente collegate, questo per gestire la complessità quando il numero di classi è elevato → utili quindi per dividere le varie funzionalità in parti diversi nel caso ci siano molte classi da realizzare

→ I package sono usati nelle API Java per organizzare la quantità di classi presenti ad esempio in Java 8 ci sono 217 package con 4240 classi! → package che quindi permette di limitare la ricerca a ciò di cui uno ha veramente bisogno!

→ alcuni esempi del package API di java sono:

- java.io → contiene tutte le classi legate all'input/output
- java.lang → classi fondamentali del linguaggio (la classe object stà qua)
- java.lang.annotation Provides library support for the Java programming language annotation facility.
- java.lang.instrument Provides services that allow Java programming language agents to instrument programs running on the JVM.
- java.lang.invoke The java.lang.invoke package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
- java.lang.management Provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime.

- `java.lang.ref` Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
- `java.lang.reflect` Provides classes and interfaces for obtaining reflective information about classes and objects.
- `java.math` → provvede le classi per gestire applicazioni matematiche
- `java.net` → per accedere alle funzionalità di rete  
→ ci sono poi un sacco di package per la creazione di interfacce utente come `java.awt` → `java.awt.event` - `java.awt.geom` - `java.awt.image` ecc...

+Come usare i package??

→ un modo è usare il nome completo composto dal nome del package e il nome della classe → ad esempio: `java.net.InetAddress ip = java.net.InetAddress.getLocalHost();`

→ la classe `InetAddress` sta appunto dentro il package `java.net`

→ oppure possiamo semplicemente usare l'import a inizio del file e scrivere:

`import java.net.InetAddress; ...`

`InetAddress ip = InetAddress.getLocalHost();`

→ così che quando facciamo riferimento a `inetAdress` facciamo riferimento a `java.net`

+Potremmo addirittura importare tutte le classi di un package scrivendo ad esempio:

`import java.net.*; ... InetAddress ip = InetAddress.getLocalHost();`

→ notare però l'organizzazione gerarchica dei package... come abbiamo visto prima per `java.awt` → se però uso l'asterisco mi vado a importare tutte le classi del package `java.awt`

→ e non anche le classi dei subpackages come `java.awt.geom` o altre... ( nel senso la classe `geom` c'è... ma non le sue sottoclassi! )

→ per cui se mi servono le classi dei subpackages → dovrò importarle esplicitamente!

+è anche possibile definire un package...

Per il nome del package di solito si usa:

– il dominio DNS rovesciato della ditta/istituzione, seguito dal nome della applicazione/libreria e quindi dal nome della sotto parte in cui è organizzata l'applicazione  
Ad esempio:

- `org.apache.commons.collections4.multimap`
- `it.unifi.myproject.db` (per le classi di gestione database)
- `it.unifi.myproject.ui` (per le classi di gestione interfaccia utente)

→ notare appunto che `unifi.it` è stato rovesciato! → notare anche come si siano divise le classi in gestione di database da quelle di gestione interffacce dandogli un preciso nome

+Esistono però delle prescrizioni specifiche sul filesystem → I file `.java` devono essere organizzati in una struttura delle directory seguendo il nome del package → ad esempio o la classe “`it.unifi.myproject.db.ImportData`” deve trovarsi in:

`it/unifi/myproject/db/ImportData.java` !! (insieme a questa ci saranno altre classi che fanno parte dello stesso package)

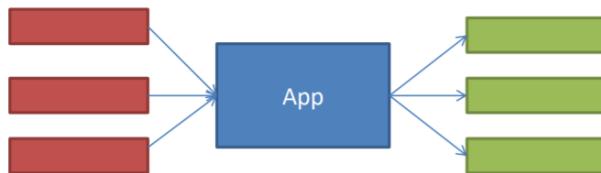
→ inoltre si deve rispettare il vincolo che all'interno del file `java` ci deve essere scritto il nome intero del package → ad esempio:

```
package it.unifi.myproject.db;
... public class ImportData ... { ...}
```

+L'avevamo già detto ma lo ripetiamo... Quando il modificatore di visibilità (public/protected/private) di una classe/attributo/metodo è omesso la visibilità è package-private cioè è visibile solo alle classi che fanno parte dello stesso package.  
→ se dunque le classi sono dello stesso package, non c'è bisogno di fare nessun import! → classi dello stesso package sono automaticamente visibili tra di loro

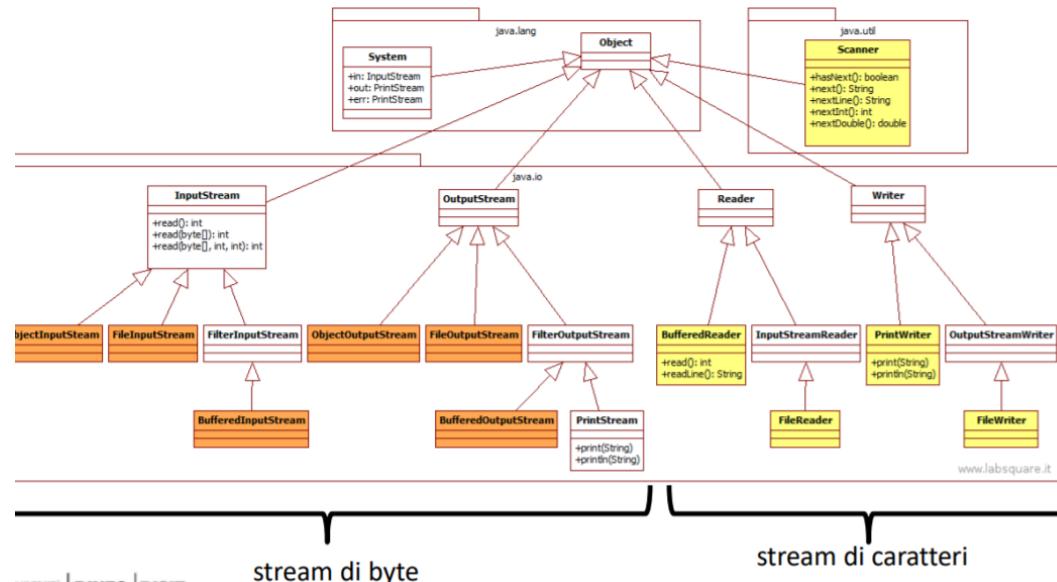
## Streams

Per accedere a dei dati in ingresso si usano degli stream input, mentre per dare un risultato si usano gli stream di output!



→ applicazione che legge da stream di input e produce dati da degli stream di output!! (che non è solo quello a schermo, ma possono essere anche dei file!)

I dati vengono prelevati (pull) da input stream e inviati (push) a output stream → l'applicazione quindi ricava gli input con particolare interfacce e produce gli stream di output  
→ Java fornisce un insieme di classi per la gestione degli stream (simili al cin e cout del C++) → e anche qui esiste una gerarchia all'interno del package



→ sono principalmente divisi in 2 → una parte che mi serve a manipolare e accedere a stream di byte, l'altra invece formata da classi che permettono di gestire stream di caratteri  
→ diversi appunto dai byte in quanto sono a 16 bit e seguono il formato UNICODE  
→ notare che tutte le classi derivano dalla classe object → classe da cui derivano tutti gli oggetti (e appunto fa parte del package java.lang)  
→ la classe system è proprio quella che abbiamo visto finora quando abbiamo usato la println! → ha quindi gli attributi inputstream e printstream che permettono rispettivamente di

leggere/scrivere qualcosa → inputstream e printstream che in realtà sono due classi e sono presenti nello schema! (in particolare printstream ha al suo interno i metodi println() e print())

+Vediamo quindi più in dettaglio la classe inputStream e cosa ci permette di fare fra le varie funzionalità!

```
FileInputStream f = new FileInputStream("file.bin"); → creo un nuovo oggetto
FileInputStream che permette di accedere al file messo fra parentesi
int b = f.read(); // legge un byte dal file e ritorna un intero
// ritorna -1 se raggiunto fine del file → infatti queste funzioni tirano fuori dei byte e dunque
può capitare di aver raggiunto la fine del file!
byte[] bb = new byte[100];
int n = f.read(bb); // legge in bb al più 100 byte → sempre usando la funzione f.read ma
stavolta leggo più di un byte alla volta! → in questo caso dunque se nel file cento caratteri →
n indica quanti letti, altrimenti se arrivo prima alla fine del file → ad esempio leggo 50
caratteri → n sarà 50, infine se non riesco a leggere niente mi ritorna -1!
int n = f.read(bb, offset, len); → versione di overloading dove si parte spostandoci di un
certo offset all'interno del buffer bb → len dice comunque quanti byte leggere dallo stream
f.close(); //chiude file → è importante ricordarsi di chiudere il file, in quanto se non lo
facciamo il sistema java, non lo rilascia... generando dei problemi → ad esempio se
possiamo aprire un numero limitato di file, questo potrebbe essere un problema!
→ sistema operativo da infatti la possibilità di accedere contemporaneamente ad un numero
limitato di file → e nel caso di applicazioni server con molti accessi, dimenticarsi di chiudere
file sarebbe un problema
```

+In modo simmetrico agisce FileOutputStream

```
FileInputStream f = new FileInputStream("file.bin");
int b = f.read(); // legge un byte dal file
// ritorna -1 se raggiunto fine del file
byte[] bb = new byte[100];
int n = f.read(bb); // legge in bb al più 100 byte
//n indica quanti letti (-1 = fine file)
int n = f.read(bb, offset, len);
f.close(); //chiude file
```

→ come prima posso decidere di scrivere solo una parte del singolo array, attraverso l'offset

+Siccome in java non esiste un distruttore → o non sappiamo insomma quando viene  
chiamato → sarà compito di queste classi chiamare la procedura di rilascio dei file → cosa  
che invece in C++ viene fatta in modo automatico con il distruttore

+Infine abbiamo il printStream che è una classe particolare → nella gerarchia è infatti una  
specializzazione della filterOutputStream → è infatti una classe che fa da intermediario  
ad esempio:

```
OutputStream os = new FileOutputStream("file.out");
```

`PrintStream p = new PrintStream(os);` → creato l'oggetto OutputStream lo passo all'oggetto printStream! → il quale fa da intermediario all'accesso di outputStream  
... `p.print("a"); p.println("hello!");` → queste stringhe vengono quindi mandate verso il stream di output e quindi porterà alla scrittura del file → ho un interfaccia semplificata! → altrimenti usando direttamente FileOutputStream, posso usare solo dei byte → per cui dovrei convertire in un stringa e poi fare la scrittura!  
→ System.out è una printStream! → che stampa su un file particolare che è l'output

Riassumendo → Le classi FilterInputStream e FilterOutputStream sono usate per filtrare uno stream di input o output mentre la classe PrintStream è derivata da FilterOutputStream (come le classi BufferedInputStream e BufferedOutputStream → che anche loro fanno da intermediario... buffered indica che il contenuto viene tenuto all'interno prima di essere inviato/letto → parte intermedia in cui si memorizza il tutto e serve per velocizzare l'accesso!)

+Altre classi ancora sono quelli di InputStreamReader → anche questi adottano il concetto del filtraggio → gli InputStreamReader funzionano anche loro da intermediari

```
InputStream is = new FileInputStream("file.txt");
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
int c = br.read() //legge un carattere (16 bit)
char[] s=new char[100]
int n = br.read(s); //legge al più 100 caratteri e
 // ritorna quanti caratteri letti
String line = br.readLine(); //legge una riga (fino a \n)
 //ritorna riga senza "a capo" o
 //null se file finito
```

→ l'InputStreamReader mi permette di leggere dei caratteri da un inputStream (che in questo caso specifico è un file di testo)  
→ è presente anche un bufferedReader al quale passo un InputStreamReader (rivedere gerarchia) che dunque mi permette di accedere in modo bufferizzato al file → e tramite questo riesco a leggere i caratteri dal file.txt → notare che i caratteri sono tutti su 16 bit! (come al solito read() dà -1 se sono arrivato alla fine del file)  
→ come per gli esempi visti primi decido di prendere un serie di caratteri insieme usando un buffer!  
→ infine è presente una funzione che è in grado di leggere una riga! → la funzione readLine ritorna un stringa che viene letta fino a che non si trova il carattere \n (occhio però al fatto che lo \n non viene riportato nella stringa finale)  
(readLine credo non sia presente in InputStreamReader ma solo in bufferedReader

+Esempio di uso → con una funziona statica di copia

```

static void copy(InputStream is, OutputStream os) throws IOException {
 int c;
 while((c=is.read()) != -1){
 os.write(c);
 }
}

```

```

FileInputStream inf = new FileInputStream("c:/tmp/file.txt");
FileOutputStream outf = new FileOutputStream("c:/tmp/file-copy.txt");
copy(inf,outf);
inf.close();
outf.close();

FileInputStream inf2 = new FileInputStream("c:/tmp/file-copy.txt");
copy(inf2, System.out);
inf2.close();

```

→ notare che passo come parametri inputStream e outputStream → potrò quindi passare anche qualsiasi cosa sia derivata da queste 2 classi

→ insostanza la funzione copy legge tutti i byte fino alla fine del file, e li scrive nel file di output!

→ notare anche che alla fine chiudo i 2 file!

→ alla fine riuso lo stesso file (creando prima un nuovo riferimento a FileInputStream) → e faccio la copia sul file di System.out → ovvero un file di printStream! (che è derivata da outputStream) → per stampare a schermo il contenuto del file

→ notare infine che il metodo appena definito può lanciare un'eccezione! (occhio se volessi provare sul compilatore il programma vanno importate sia i package di (input/output)stream che IOException → sono tutte di java.io) → l'eccezione viene lanciata proprio nel caso in cui il file di cui cerchiamo di fare la copia, non esiste o non viene trovato!

+ Come detto prima, nel caso manchi \n è un problema per stampare a schermo

→ in questo caso ad esempio dopo "questo è un file" non sono andato a capo! → e quindi continuando la ricerca del \n e non trovandola non la stampa (rimane bufferizzato)

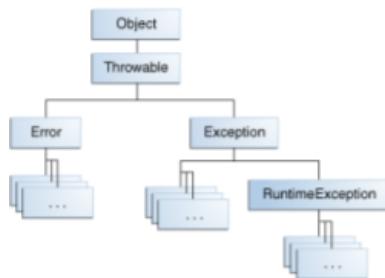
→ come risolvere il problema?? → uso il metodo os.flush() → che manda comunque tutto verso l'output → anche ciò che è rimasto bufferizzato

- +Pezzetto sui problemi della decodifica → per colpa di un sacco di codifiche strane dei testi dovuti ai vari linguaggi(arabo, cinese, ecc...) , al testo in realtà è associato anche l'encoding → ovvero come quel testo è codificato...
- la codifica UTF-8 è la codifica multibyte dell'unicode → dove per un carattere non abbiamo una lunghezza fissa, ma variabile → ad esempio un carattere giapponese può essere codificate con 2-3 byte, mentre quelle del nostro alfabeto sono codificate con un byte solo ! (il bellini ci ha battuto molte volte la testa sulle codifiche di un testo...)

## Eccezioni

- Le eccezioni vengono lanciate (throw) durante l'esecuzione del programma per indicare una condizione errata che non può essere risolta (in quel momento) → “ si dice non posso fare niente, beh lanciamo un eccezione”
- Le eccezioni possono essere gestite per risolvere il problema o per dare indicazione di errore e continuare con operazione successiva → oppure ancora terminare l'intero programma se necessario (cosa meglio da evitare) → di fatto se Le eccezioni se **non** gestite (catch) bloccano l'esecuzione dell'intero programma !

+Anche le eccezioni seguono il modello gerarchico all'interno di un package



- sotto la classe fondazionale `object` è presente la classe `Throwable` → che rappresenta la classe degli oggetti che possono essere lanciate → classe che è successivamente divisa in errori e eccezioni → quest'ultime sono nei casi di situazioni di errore conosciute, prevedibili e documentate (ad esempio `FileNotFoundException`)
- mentre gli errori, sono qualcosa di imprevedibile, generati nell'uso della classe → ad esempio `IOError` → errore di basso livello e non prevedibile...(ad esempio un errore del disco)
- le eccezioni sono suddivise a loro volta in `RuntimeException` → Sono errori imprevedibili dovuti ad un errore logico nella **applicazione** (es. `NullPointerException`)
- gli errori di solito sono più legati all'hardware/sistema operativo, mentre le eccezioni legate alle applicazioni

+Come gestire le eccezioni?

|                            |                                                                                                                                                                                                             |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>try</b> {               |                                                                                                                                                                                                             |
| ... istruzioni...          | in queste istruzioni o nei metodi richiamati<br>può essere lanciata una eccezione                                                                                                                           |
| }                          |                                                                                                                                                                                                             |
| <b>catch(Classe exc)</b> { | queste istruzioni gestiscono una<br>eccezione di tipo Classe o una sua<br>derivata, si possono avere più sezioni<br>catch su eccezioni diverse                                                              |
| ... istruzioni...          |                                                                                                                                                                                                             |
| }                          |                                                                                                                                                                                                             |
| <b>[finally</b> {          | queste istruzioni eseguite comunque<br>sia che venga venga lanciata eccezione<br>gestita che non gestita. Serve a<br>rilasciare risorse eventualmente<br>acquisite di cui si deve garantire il<br>rilascio. |
| ... istruzioni ...         |                                                                                                                                                                                                             |
| }]                         |                                                                                                                                                                                                             |

→ rispetto al C++ è presente un ulteriore sezione finally (opzionale) dove si mettono le istruzioni che comunque devono essere eseguite anche se un'eccezione non gestita può portare al termine del programma → come scritto si usa nel caso eventuale di rilasciare risorse acquisite (all'interno del try) → ad esempio come abbiamo visto prima dobbiamo necessariamente, dopo aver usato un certo file, chiuderlo → altrimenti l'handle del file non è più utilizzabile...

## +Esempio

```
try {
 int[] v = new int[10];
 ...
 v[10] = 10; → Genera eccezione
 ...
} catch(IOException e){
 e.printStackTrace();
} catch(Exception e){ → Gestisce l'eccezione
 e.printStackTrace();
 System.out.println("msg:" +e.getMessage());
} finally {
 System.out.println("finally!");
```

→ notare che si possono avere più sezioni catch, e ovviamente sarà attivata solo quella che ha una certa relazione con l'eccezione lanciata (in questo esempio l'eccezione lanciata è catturata dal secondo blocco catch → più generale)

→ occhio ad essere esplicativi nel gestire un'eccezione, altrimenti questa rimane nascosta e non ci accorge del problema

→ la funzione printStackTrace è molto interessante → in quanto fà vedere tutto il percorso di come viene lanciata un'eccezione e il tipo di questa → il print... dunque non fà altro che stampare a schermo quello che abbiamo detto, ma non blocca il programma! → dopo questa infatti si passa all'istruzione successiva

→ se invece non facciamo niente e non siamo esplicativi nel blocco catch (come detto prima) non sappiamo che è stata lanciata un'eccezione e dunque è presente un problema

+Le eccezioni che non sono Error o RuntimeException ( dette checked exceptions → in qualche modo prevedibili) devono essere gestite o dichiarate nella firma del metodo/costruttore che possono essere lanciate (throws), il chiamante dovrà gestirle o

dichiararle nel suo throws → è un obbligo del linguaggio da rispettare!! (nel C++ invece no!) ad esempio:

```
class ... { ... public void method(...) throws ...lista eccezioni... { ... } }
```

→ chi userà quindi method() o dovrà gestire l'eccezione, oppure ne lancerà un'altra → passando comunque l'obbligo di controllarla ad altri!

→ Se non viene fatto il compilatore genera errore!

+Esempio:

```
class EsempioInputStream {
 int chkFile() throws IOException {
 InputStream is=new FileInputStream("input.bin");
 int c=0, chk=0;
 try {
 while((c=is.read()) != -1){
 chk ^=c; //xor bit-wise
 }
 } finally {
 is.close();
 }
 System.out.println("chk: "+chk);
 return chk;
 }
}
```

→ notare la presenza di finally → dove chiudo il file

→ notare anche che non è presente un blocco catch! → domando a chi userà il metodo chkFile() il successivo controllo del catch (se addirittura nemmeno lui lo controlla sarà il main che lo lancia, e poi la JVM lo gestisce e dice che c'è stata un exception)

+Stesso esempio ma in versione alternativa:

```
class EsempioInputStream {
 int chkFile() throws IOException {
 InputStream is;
 try {
 is = new FileInputStream("input.bin");
 } catch(FileNotFoundException ex) {
 is = new FileInputStream("input2.bin");
 }
 int c, chk=0;
 try {
 while((c=is.read()) != -1) chk ^=c;
 } finally {
 is.close();
 }
 System.out.println("chk: "+chk);
 return chk;
 }
}
```

→ se non è presente il file input.bin prova ad aprire input2.bin → e se nemmeno questo c'è  
→ genera anche lui l'eccezione FileNotFoundException → che però stavolta verrà gestita da chi la chiama in quanto viene dichiarato throws IOException

+ Può anche capitare di lanciare esplicitamente un'eccezione → sempre con il throw, ad esempio: **throw new IllegalArgumentException("parametro errato");**  
→ L'oggetto lanciato deve essere di una sottoclasse di Throwable → che è la radice della gerarchia delle eccezioni!

+ Si possono definire nuove eccezioni, definendo una sottoclasse di Exception o RuntimeException → Per comprensibilità il nome della classe dovrebbe terminare in Exception → infine Il costruttore può prendere come parametro il messaggio associato (se necessario) → ad esempio:

```
public class MyException extends Exception {
 public MyException() { super(); } → costruttore di default → non prendo nessun messaggio
 public MyException(String msg) { super(msg); } → versione con il messaggio
}
```

→ nel momento in cui la **lancierò** scrivo **throw new MyException("errore...")** → siccome non ho derivato la nuova eccezione da un RuntimeException → questa sarà una checked exception dunque v'è gestita! → blocco catch ecc...

### Input da console

→ abbiamo visto come fare l'output, dunque come fare l'input in java??  
→ usando la stream → system.in permette di leggere dalla riga di comando del programma (risulta molto complicato purtroppo :( )  
→ è stata quindi definita la classe scanner che facilita l'interazione con la system.in, ad esempio: import java.util.Scanner; ...

```
Scanner s = new Scanner(System.in); → fà quindi da intermediario
```

```
int x = s.nextInt(); //legge un intero
float f = s.nextFloat(); //legge un float
String str = s.nextLine(); //legge una riga
```

→ classe scanner da utilizzare nel caso di input da console! (occhio però che non fà parte del package di java.io ma di java.util!)

+ Esempio:

```
Scanner s=new Scanner(System.in); int x;
do { try { System.out.print("x [1-5]: "); x=s.nextInt(); }
catch(InputMismatchException e) { x=0; → infatti non appartiene a [1,5] e riparto
s.nextLine(); //estrae l'input errato System.out.println("input non valido"); }
} while (x<1 || x>5);
```

→ il programma dunque va avanti fino a quando non inserisco un valore fra 1 e 5!  
→ la funzione nextInt chiede in input un numero intero → per cui nel caso non metta un numero intero, viene generata un'eccezione → con la funzione nextLine il valore rifiutato viene rimesso nello stream di input → se infatti non viene tolto questo input dallo scanner, continua a generare un'eccezione → se non fosse presente questo metodo, la funzione ciclerebbe all'infinito!!

→ ad esempio se immettessi “a” come input da console → se non eseguo la funzione nextLine, l’input resta nello stream, e tutte le volte con il nextInt si rimette lo stesso input generando l’eccezione!

## Foreach

In java versione 5 è stato introdotto un nuovo costrutto che sostituisce il for classico → Permette di iterare su elementi di un array o di una collection, ad esempio:

```
float[] v = new float[n];
...
for(float x : v) {
 System.out.println(x);
}
for(int i=0; i<v.length; i++) {
 float x = v[i];
 System.out.println(x);
}
```

→ i 2 codici sono analoghi fra loro → oltre a rendere più asciutto il linguaggio, questa sintassi si rende più utile nel caso di una collection

## Tipi base e Object

+Per ogni tipo di base è presente una classe derivata da Object (detta wrapper class) che permette di utilizzare classi generiche per memorizzare sia oggetti che tipi di base (in sostanza i tipi base non sono derivati dalla classe object! —> per far sì che lo siano uso delle classi particolari!)

→ ad esempio la classe int ha associato la classe Integer, allo stesso modo byte → Byte, float → Float, boolean → Boolean ecc... posso quindi rappresentare i tipi base come degli oggetti! → preso ad esempio un lista di object → ci voglio mettere dentro degli interi e li trasformo in degli Integer → a quel punto li potro mettere nella lista!

→ Il compilatore automaticamente trasforma un tipo di base in una istanza della wrapper class corrispondente (Boxing) e viceversa (Unboxing), ad esempio:

```
Integer x = 3; //boxing
Integer x = new Integer(3);
int y = x*2; //unboxing
int y = x.intValue() * 2;
```

la prima quindi crea automaticamente, un nuovo oggetto integer e lo inizializza con il valore 3, a quel punto farà anche l’unboxing → tira fuori dalla scatola il valore dell’oggetto per moltiplicatore per 2 → il compilatore infatti automaticamente si accorge del riferimento a x e fa l’unboxing

# 26/03/2021

## Java Collection Framework

Insieme di interfacce e classi per la gestione di insiemi di oggetti → permettono quindi di aggregare un insieme di oggetti → sono le strutture dati viste tipicamente ad algoritmi → liste - mappe - code → implementate in vari modi → come tabelle hash oppure alberi binari di ricerca → ad oggi si usano con le estensione dei *generics* (introdotti in java 5) → con simili ai template del C++ (di sintassi simile ma hanno una diversa implementazione)

→ ad esempio: "List<Type> x = new ArrayList<>();" → x è una lista che contiene un certo tipo → List è in realtà un interfaccia che fornisce una serie di funzionalità (come inserimento, oppure lettura della lista) e può essere assegnato con diversi tipi → ad esempio ArrayList oppure "List<Type> x = new LinkedList<>();" → lista implementata con una lista concatenata (invece che array di liste) → diverse caratteristiche per le performance! array di list migliori per cercare un elemento, mentre linkedlist migliori nel caso dovessimo aggiungere spesso nuovi elementi!

→ se infatti andassimo a Vedere come è implementata la funzione .get() → Per gli ArrayList è semplicemente una return con l'indice mentre per le LinkedList list è molto più complicata!

→ occhio però che possiamo mettere come Type una classe derivata da object e non un tipo base! → per fare la stessa cosa di una lista di interi → va usato integer!

→ per accedere agli elementi di queste collezioni si usano gli *iteratori* → oppure il costrutto for each visto prima!

+Esempio:

```
import java.util.LinkedList; //classe concreta
import java.util.List; //interfaccia ...
List<Integer> lst = new LinkedList<>(); → lst lista di interi usata una lista linkata(si fa riferimento ad una linkedList che implementa l'interfaccia list)
→ mettere "<>" mi fa evitare di dover riscrivere integer
lst.add(1); lst.add(2); lst.add(3); → notare che si usa il boxing → l'intero 1 viene trasformato in un integer e poi aggiunto alla lista!
System.out.println(lst); // [1, 2, 3]
lst.remove(1); //rimuove elemento in posizione 1 (parte dall'indice zero → quindi toglie 2!)
for(int y: lst) { System.out.println(y); } → uso del foreach → prendo i valori(facendo l'unboxing) e li metto nella variabile y per poi essere stampati!
```

+Se volessi fare la stessa cosa del foreach ma usando gli iteratori (occhio a inserire la classe degli iteratori)

```
Iterator<Integer> i = lst.iterator();
while(i.hasNext()) { int y = i.next(); System.out.println(y); } → esegue fino a che ha un elemento successivo! → simile ai ptr → next punta sempre all'elemento successivo
+Se vado oltre le dimensioni della lista → pam! → eccezione!
→ ovviamente posso iterare semplicemente con un ciclo for → si sfruttano i metodi .size() e .get(indice) per ottenere lunghezza della liste e l'elemento dell'i-esima posizione!
```

fine java

## Gestione dei processi

Un sistema operativo esegue una varietà di programmi che sono condivisi in 2 tipologie:

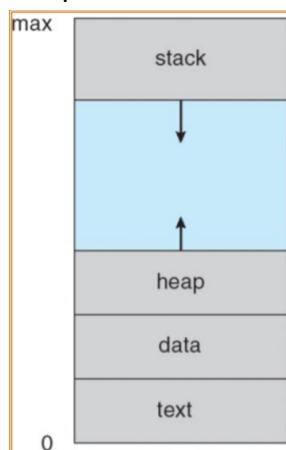
- Sistemi batch → sistemi che hanno da elaborare molti dati e sono fatti da dei sistemi che operano su dei job che devono essere effettuati per fare analisi... cose che non dovranno essere necessariamente interattive ma dovranno eseguire dei conti → ad esempio un sistema bancario → programmi che quindi leggono informazioni dal database → le elaborano e producono dei nuovi dati senza necessariamente l'interazione con l'utente
- Sistemi in time-sharing → sono i sistemi usati dagli utente direttamente i quali forniscono dei task che dovranno essere eseguiti e ogni utente vuole vedere che il proprio programma continui il lavoro richiesto → dunque fornendo determinati output a seconda degli input dell'utente

→ sono comunque entrambi dei sistemi che condividono l'uso del calcolatore con il pc

+dietro un *processo*, c'è comunque un programma che deve essere eseguito → dunque per diversi processi diversi programmi a seconda dei sistemi!

→ un processo è quindi un programma in esecuzione e l'esecuzione del programma deve procedere in modo sequenziale → dunque seguendo una serie di istruzioni

→ userà poi determinate chiamate di sistema per interagire con il S.O e fare quello che deve fare → in genere un processo include: program counter - stack e una sezione dati (in più una parte del codice da eseguire)

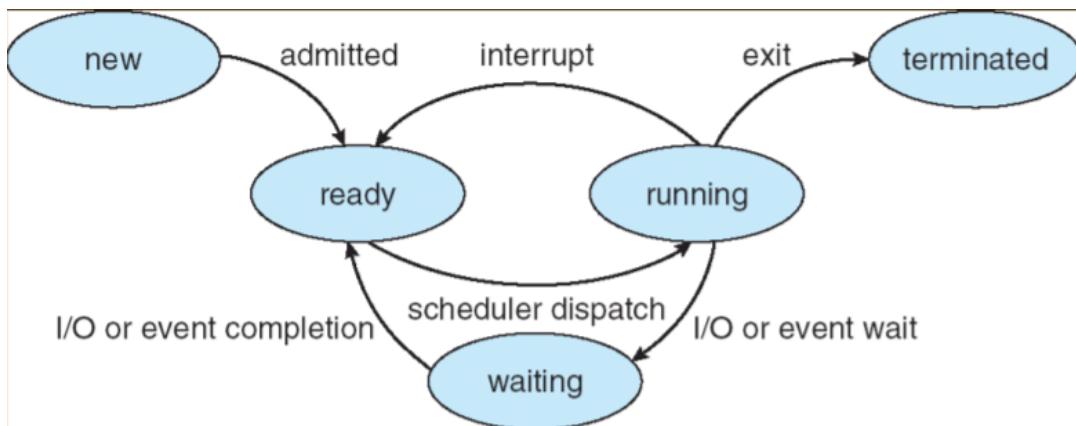


→ il processo in memoria può essere visto come una parte che contiene lo stack, un'altra che contiene lo heap(che può crescere nel tempo, se ci sono risorse allocate dinamicamente) → lo stack invece contiene le variabili locali e i punti di ritorno delle procedure e delle funzioni (dunque cresce nel verso opposto)

→ data contiene variabili globali e altre cose belle e infine text contiene il codice che deve essere eseguito dal nostro processo!

→ questo è un processo singolo → ma in realtà se ne possono eseguire contemporaneamente più di 1 → il sistema operativo deve astrarre la gestione della memoria in modo che ogni processo vede lo spazio di memorizzazione riservato a lui e non interagisca o possa modificare degli spazi d'interazione degli altri processi!

- + Quando un processo è in esecuzione cambia il proprio stato → esiste una variabile che esplicità lo stato in cui si trova un processo... questi stati sono:
  - new: il processo è stato creato → dunque lanciato ma non è stato eseguito nessuna istruzione
  - ready: il processo è in attesa di essere assegnato alla CPU → dunque dopo aver allocato la memoria necessaria e pronto all'esecuzione
  - running: istruzioni del processo sono in esecuzione
  - waiting: il processo è in attesa di un evento → può lasciare la CPU per essere usata da altri processi
  - terminated: il processo ha finito l'esecuzione → e tutte le risorse che sono state acquisite sono state rilasciate



- diagramma di transizione degli stati → notare quindi che si rimane nello stato running, finché o arriva una richiesta di input/output → ovvero una chiamata di sistema che chiederà ad esempio che vuole leggere qualcosa da un file → il processo entra nello stato di waiting → nel quale altri processi potranno diventare da ready a running!
- aspettando che la richiesta venga realizzata rimane in questo stato → quindi completato questo il processo torna nello stato ready, pronto all'esecuzione
- nello stato ready c'è quindi una serie di processi che saranno tutti quelli pronti per essere eseguiti → nel caso in cui si abbia un sistema composto da più CPU allora potremmo eseguire più processi contemporaneamente → per cui anche nello stato running potrebbero esserci più processi
- se dunque arriva l'istruzione di uscita si passa allo stato terminated
- si nota però dal diagramma che può capitare anche di passare dallo stato running direttamente al ready → processo in esecuzione torna ad essere pronto → ad esempio nel caso di un interrupt → viene tolta la CPU e si torna allo stato di ready

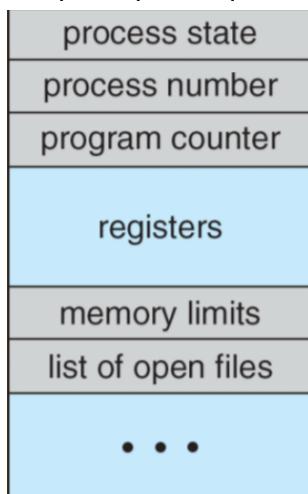
## Process control Block(PCB)

- Una struttura dati del sistema operativo (del kernel) che permette di tenere tutte le caratteristiche del processo → prima di tutto quindi risulta contenuto lo *stato del processo*
- in quale stato si trova tra quelli che abbiamo visto
- successivamente è presente il *process number* → (PID → process identifier) che serve a identificare univocamente il processo all'interno del sistema

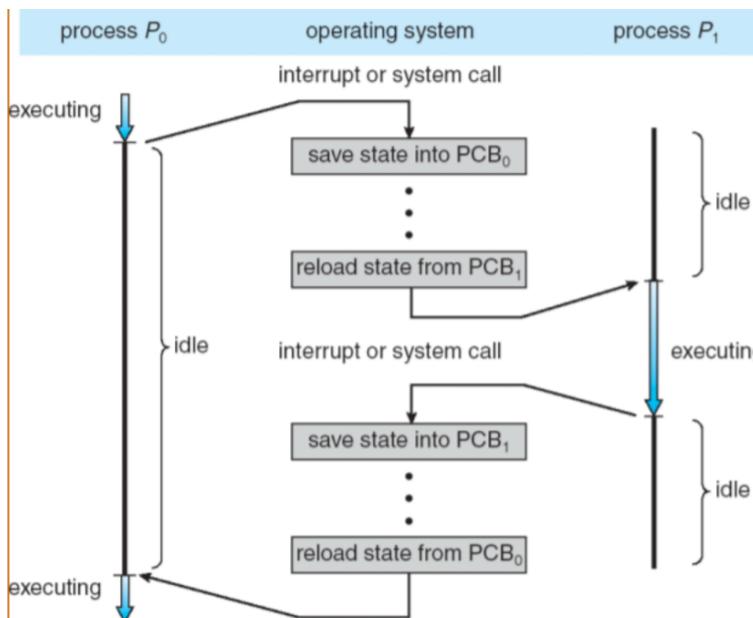
→ Poi abbiamo il program counter → all'interno del PCB viene infatti contenuto il valore del program counter → viene fatto questo però quando si passa dallo stato di running a quello di waiting oppure ready, in modo da salvare la posizione raggiunta nel program counter e quindi poter riprendere l'esecuzione dal punto in cui ci era interrotto!

→ ovviamente questo non verrà aggiornato nello stato running ma solo nei casi detti prima  
Oltre a questo abbiamo quindi dei registri → anche questi utili per mantenere lo stato del programma → poi sono presenti informazioni sulla memoria(zona di memoria associate al processo in esecuzione) → Informazioni per lo scheduling della CPU (quanta CPU usata fino a quel momento...) → Informazioni di accounting → ovvero quante risorse sono state utilizzate (per potere fare successivamente una sorta di pagamento alla CPU ??)

→ poi ancora Informazioni sullo stato dell'I/O → ad esempio all'interno del PCB, si riescono a sapere quali e quanti file sono stati aperti/utilizzati da un processo ecc...



+Quando dunque la cpu passa, l'esecuzione da un processo ad un altro vengono fatte una serie di azioni:



→ il processo P0 viene ad un certo punto interrotto → si salva lo stato nel PCB,  
successivamente il sistema operativo decide di far entrare in esecuzione il processo P1,

quindi carica lo stato di questo processo nella CPU, ripristinando tutti i valori dei registri e anche il Program counter nel punto in cui era arrivato → quindi passa all'esecuzione del processo P1 → il quale può terminare oppure venire bloccato da un'interruzione → dunque si risalva lo stato dalla CPU al PCB e infine viene ripristinato il processo P0 che tornerà all'esecuzione!

→ tipicamente non è così semplice! → quando abbiamo n processi → si passerà dalle esecuzioni dei processi P1, P2, P3 e così via fino a tornare a P0

→ comunque nel PCB ci saranno sempre tutte le info per tenere lo stato corrente del processo → per dunque salvarle e poterle successivamente riutilizzarle!

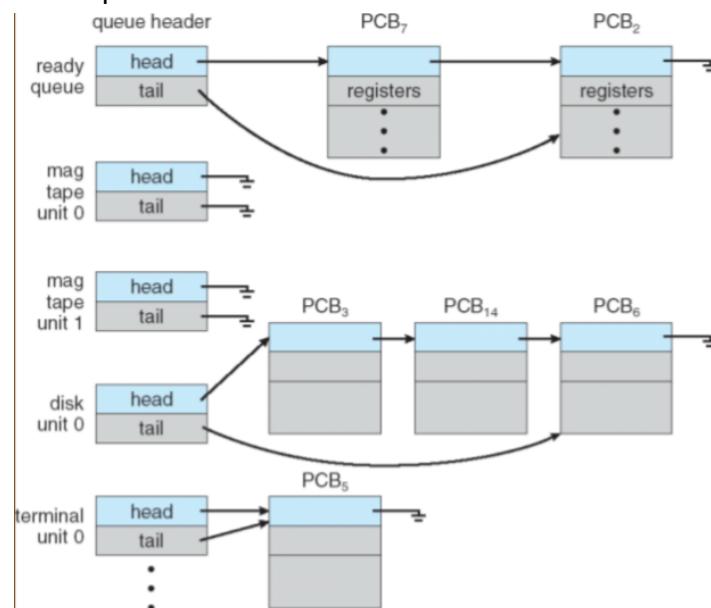
### Code per lo scheduling dei processi

Tipicamente sono presenti 3 tipi di code:

- Job queue – insieme di tutti i processi del sistema → processi che devono essere eseguiti sul sistema (tipica dei sistemi batch)
- Ready queue – insieme di tutti i processi che risiedono in memoria pronti e in attesa di essere eseguiti → dunque tutti i processi che si trovano nello stato ready
- Device queues – insieme di processi in attesa di un I/O su device

→ il sistema operativo fa migrare i processi dalle varie code → nel momento iniziale entrano nella job queue → poi quando c'è abbastanza memoria per caricare il processo vengono messe nella ready queue → coda dalla quale entreranno successivamente in esecuzione → infine quindi andranno nelle code dei device per fare l'input/output richiesta → finito il processo ritorna alla ready queue fino al termine

+Esempio:



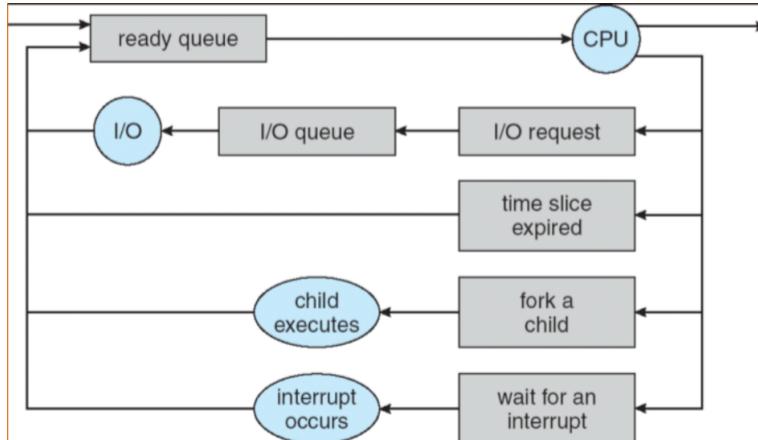
→ in questo caso i processi vengono tenuti in delle liste → un modo è quindi quello di mettere all'interno del PCB che serve a tenere tutti gli elementi che fanno parte della lista (che rappresenta la coda) in cui il processo è contenuto! → questo può essere fatto in quanto i processi o si trovano nella ready o nei device queue ma non in entrambe!

→ notare i puntatori sia alla testa della lista che alla coda

+è presente una device queue per ogni dispositivo nel quale il processo può essere in attesa... → nell'esempio ci sono 2 nastri magnetici senza nessun processo in coda, poi un disco con dei processi che ad esempio sono in attesa di ricevere dei dati dal disco (processi 7 e 2 invece sono in attesa della cpu)

→ infine il processo 5 è ad esempio in attesa che l'utente mandi un certo input!

+un modo per rappresentare i processi è il seguente:



→ dalla ready queue i processi vanno nella CPU → dove possono o uscire terminando il programma, oppure se viene fatta una richiesta di input/output → andrà verso una coda I/O per poi eseguire la richiesta voluta e dunque tornare nella ready queue

→ oppure dalla cpu esce e torna alla ready queue poiché il tempo a disposizione è finito

→ oppure si crea un nuovo processo → fork a child → a questo punto si esegue il processo figlio, oppure infine si aspetta un'interruzione e quando a questa si torna alla ready queue!

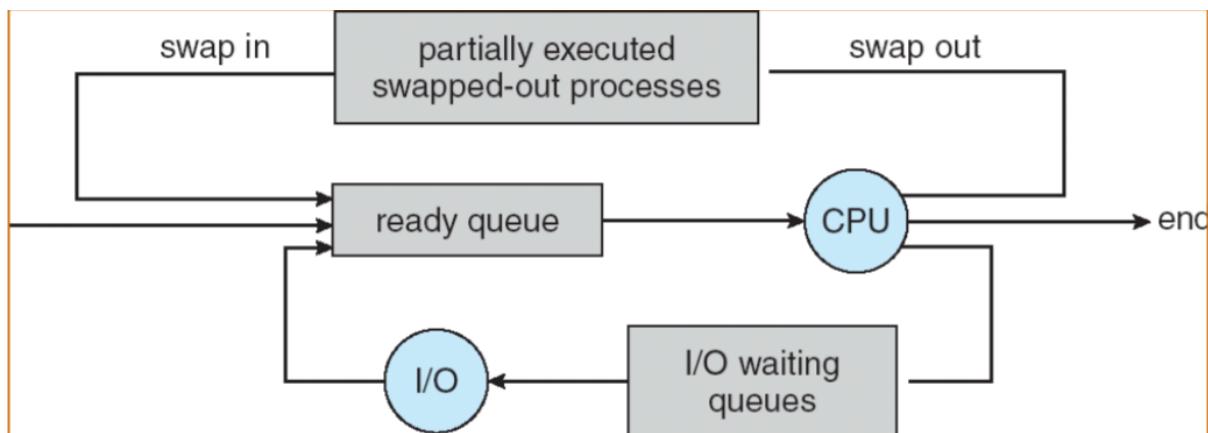
## Schedulers

- Scheduler a lungo termine (o job scheduler) – in un sistema batch seleziona quali processi tra quelli in attesa devono essere portati nella ready queue → sono quindi in attesa proprio di essere portati ad esecuzione
- Scheduler a breve termine (o CPU scheduler) – seleziona quale processo deve essere eseguito e allocato sulla CPU (tra quelli ready) → il processo ha ad esempio caricato tutta la memoria, ma non c'è una CPU disponibile (oppure questa non è disponibile)
  - nei calcolatori più semplici avremmo a che fare con lo scheduler a breve termine che decide di portare in esecuzione i vari processi che sono attivi
  - scheduler a lungo termine invece tiene conto anche dei vincoli → ad esempio periodici → ogni giorno esegui un certo processo messo in coda...

+Esiste in realtà anche un scheduling a breve termine → Serve ad eliminare dalla memoria processi parzialmente eseguiti riducendo il grado di multiprogrammazione

→ cioè quando il sistema si accorge che troppi processi in esecuzione portano a dei problemi → il sistema operativo può decidere di portare fuori un processo dalla CPU o addirittura portarlo fuori dalla memoria → questo se il processo usa troppa memoria e dunque sta creando problemi sull'organizzazione della memoria

→ il processo in questione verrà quindi tolto dalla memoria (swap out), ne salvo lo stato di questo così da poterlo successivamente riutilizzare!(swap in)



→ scheduling a breve termine fà fare questo! → fà decidere frà quelli che sono nella ready queue quale portare alla cpu → ci sono vari criteri per decidere quali processi portare in esecuzione → ad esempio un modo è ordinali e metterli in una coda FIFO

→ in quello a medio termine invece il S.O può decidere di non fare eseguire un determinato processo → il quale viene fermato sul disco → la memoria usata da un processo viene eliminata e messa a disposizione di altri processi → lo scheduling a medio termine dunque decide quando intervenire e cosa fare scegliendo sia tra quelli in esecuzione oppure tra i ready quale non fare più eseguire tirandolo via!

- Scheduler a breve termine è invocato molto frequentemente (millisecondi) → deve essere veloce a prendere una decisione → se troppo lungo infatti può essere un spreco di memoria da parte della CPU decidere quale portare via
- Scheduler a lungo termine è invocato di rado (secondi, minuti) → può essere lento → posso quindi dare più tempo alla CPU per fare questa decisione (che è fatta poche volte) → Lo scheduler di lungo termine controlla il grado di multiprogrammazione → ovvero quanti processi sono in esecuzione sul sistema!
- I processi possono essere di 2 tipi:
  - A prevalenza di I/O (I/O-bound) – passa più tempo a fare I/O che computazioni → ad esempio accedono a file - database → aspettando molto spesso dei feedback dell'I/O → un esempio pratico è quando aspettano l'input dell'utente → I/O bound interattivo → maggior parte del tempo sono in attesa
  - A prevalenza di CPU (CPU-bound) – spende più tempo a fare computazioni → processi che hanno già tutti i dati in memoria e quindi devono fare operazioni su questi che comportano notevoli costi di CPU

### Cambio di contesto

Quando la CPU viene assegnata ad un altro processo, il sistema deve salvare lo stato del processo e caricare lo stato del nuovo processo → questa operazione è detta *context-switch*. Il cambio di contesto comporta però un calo delle prestazioni → il sistema non fa nessun lavoro utile alla computazione mentre effettua il cambio di contesto → serve infatti solo a cambiare l'esecuzione da un processo ad un altro → deve quindi essere fatto nel modo più veloce possibile! → anche se il tempo impiegato dipende dal supporto hardware → a

seconda che la CPU abbia istruzioni specifiche per salvare lo stato della CPU (program counter e registri della cpu) in modo diretto senza dover essere altre istruzioni !!

+Ipotizziamo quindi un contesto in cui abbiamo una solo CPU, ma più processsi da eseguire contemporaneamente → abbiamo una serie di processi CPU-bound → che non fanno I/O → se li faccio eseguire uno dopo l'altro ottengo un certo tempo di esecuzione...

se invece inizio a fare dei context-switch uno dopo l'altro → i tempi dovuti a questo sono tutti tempi in più che la CPU spende → in quanto non si esegue più istruzioni del processo, ma comandi del sistema operativo! → il tempo speso per context-switch è tempo che non serve! → se invece ho processi I/O bound, sfruttare i tempi d'attesa per l'I/O, risulta importante! → se non faccio in quel momento il context switch → in quel caso dovrei aspettare che l'I/O finisca e dunque portare a spreco di tempo per la CPU!!

### Creazione dei processi

Processo genitore (parent) crea processi figli (children), che a loro volta creano altri processi → formando un albero di processi (process tree) → i processi sono legati uno all'altro attraverso la relazione di creazione → che appunto dice quali sono i processi figli di altri processi

+Riguardo la **condivisione** delle risorse (es. file aperti), si hanno 3 possibilità:

- Processo genitore e processi figli condividono tutte le risorse → ciò che è del padre è anche del figlio
- I processi figli condividono con i genitori alcune risorse
- Processi genitori e figli non condividono nessuna risorsa

+Riguardo invece l'esecuzione, si hanno 2 possibilità:

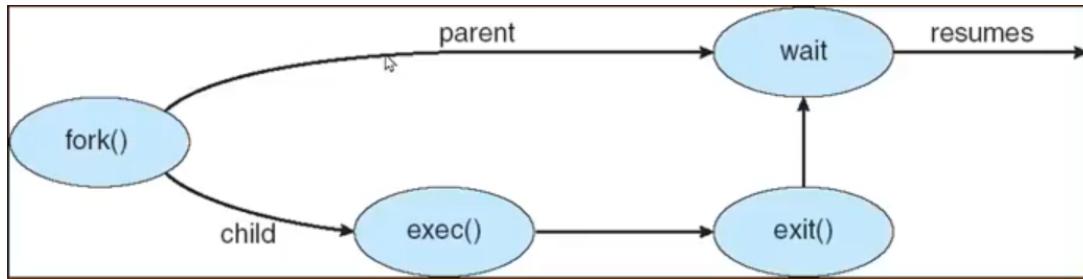
- Processo genitore e figli sono eseguiti in concorrenza → padre che lancia il figlio ma continua la sua esecuzione
- Processo genitore aspetta fino alla terminazione dei processi figli  
→ in un caso il processo genitore crea un figlio, e creato questo, entrambi poi procedono nell'esecuzione, nell'altro caso il processo genitore crea il figlio e si mette in attesa finché questo termini! (in generale si esegue la prima → aggiungendo la possibilità di aspettare la terminazione del figlio)

+Riguardo invece lo Spazio indirizzi (memoria), ho due possibilità:

- Processo figlio ha un duplicato del processo padre
- Processo figlio creato con un programma caricato già dentro  
→ nella prima possibilità in realtà il processo che è un clone di se stesso (figlio clone del padre) → nell'altra possibilità invece il padre vuole creare un figlio che deve eseguire un certo programma scritto su un certo file che specifico

→ UNIX adotta la prima politica nel quale viene prima fatta la chiamata di sistema *fork* → che crea un nuovo processo duplicato del padre → l'unica cosa che cambia tra padre e figlio è il valore ritornato da *fork* → dunque il padre ritorna id del processo figlio mentre nel figlio ritorna 0 (per capire se siamo in esecuzione nel padre o nel figlio)

→ quindi è presente la chiamata di sistema exec (usata dopo una fork) che sostituisce lo spazio di memoria con quello di un nuovo programma → il sistema operativo passa da eseguire un certo processo ad un altro che viene specificato



→ il parent aspetta fino al wait, infatti per la biforcazione il processo inizialmente cambia entità → ovvero cambia cosa sta eseguendo → mettendo in esecuzione il figlio che opera fino all'exit → il quale restituisce un valore che può essere dato al padre → serve infatti a far capire che il figlio ha finito!

+esempio di programma che crea un sottoprocesso

```

#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int main()
{
 pid_t pid; /* process id */
 /* fork another process */
 pid = fork();
 if (pid < 0) { /* error occurred */
 fprintf(stderr, "Fork Failed");
 exit(-1);
 }
 else if (pid == 0) { /* child process */
 execlp("/bin/ls", "ls", NULL);
 }
 else { /* parent process */
 /* parent will wait for the child to complete */
 wait(NULL);
 printf ("Child Complete");
 exit(0);
 }
}

```

→ la variabile pid indica l'id del processo → a quel punto si invoca la fork() → che chiede al processo di duplicarsi → crea un copia di se stesso e ritorna l'id di chi ha creato → per cui se assume un valore negativo il comando è fallito (ad esempio fine della memoria)

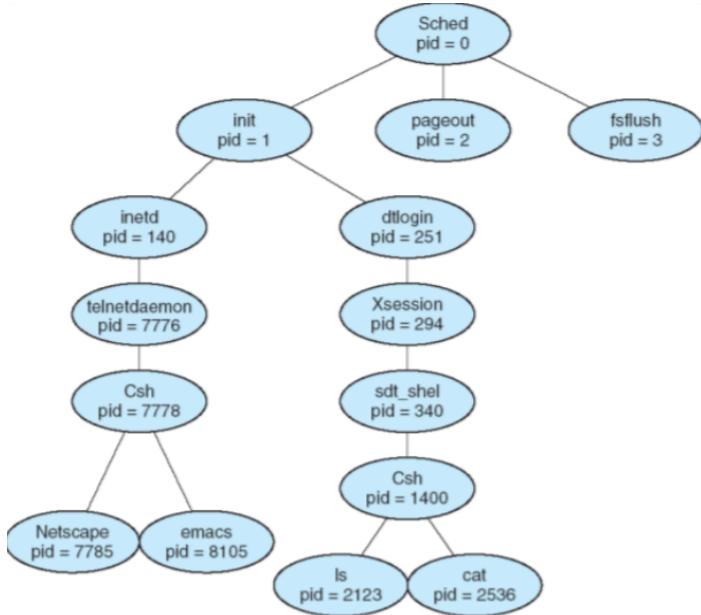
→ altrimenti se il pid = 0, vuol dire che siamo in esecuzione del figlio → che ha come valore restituito dalla pid il valore zero → per cui il processo eseguito inizialmente trasforma se stesso nell'esecuzione di un altro processo → in questo caso è quello di "ls" → che fa l'elenco dei file del filesystem

→ altrimenti ancora nell'ultimo else il padre aspetta che il figlio termini (in questo caso mette null → ignorando il risultato del figlio → anche se potrei salvare il risultato del processo figlio da qualche parte e darlo al wait)

→ quando quindi il processo figlio termina stamperà la stringa e poi esce!

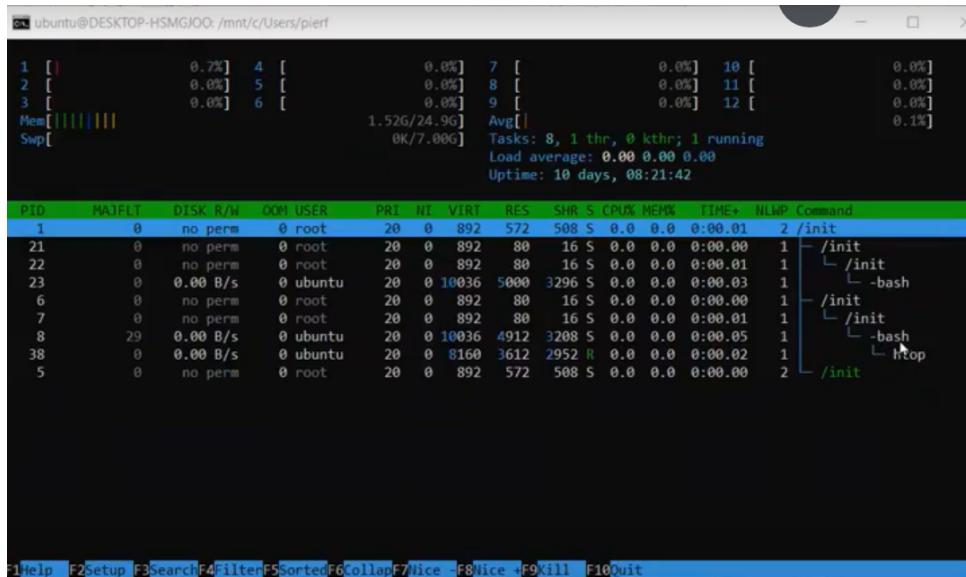
→ nel caso del padre quindi ripetiamo che si mette in attesa della terminazione del figlio! → nel caso avessi messo alcune istruzioni prima della wait allora queste agiscono in maniera **concorrente** al processo figlio! → finite queste, dovrò comunque aspettare anche la fine dell'elenco di tutti file(processo figlio)

+Esempio di albero dei processi(da solaris)



→ “cat” usato per guardare il contenuto di un file → Csh esegue una shell → dunque una riga di comando dal programma

+Esempio su una shell unix(macchina virtuale)



→ si è richiesto il comando htop → e dall'init iniziale si eseguono 2 fork → una esegue la bash, mentre l'altro esegue l'htop → e sono uno il figlio dell'altro! (in realtà il prof ha fatto 2 shell per questo motivo sono divise uno dall'altro! → se ne apro un'altra ne avrò un'altra biforcazione)

## Terminazione dei processi

Un processo quando esegue la sua ultima istruzione → chiede al sistema di terminarsi → dunque fa una **exit** → questo fornisce il valore risultato al processo padre che lo riceve dalla wait → nel C il valore intero che viene ritornato alla fine del programma → viene passato ad una chiamata di sistema exit nel momento in cui termina!

- Le risorse del processo sono rilasciate dal sistema operativo
- + Un processo genitore può terminare l'esecuzione dei processi figli → chiamando la funzione **abort** → che in realtà è una chiamata di sistema... dunque viene eseguito:
  - Se il compito assegnato al figlio non è più richiesto
  - Se il processo padre termina(prima che il processo figlio abbia terminato) → Alcuni sistemi operativi non permettono ai figli di continuare se il processo padre termina!
- per cui nel caso termina il padre tutti i processi figli vengono terminati in cascata
- Altri sistemi operativi assegnano i figli al processo iniziale → nel caso in cui termini il padre allora il figlio diventa (a sua volta) figlio del processo init
- + Processo **Zombie**: un processo **figlio** terminato ma che ha ancora pid e PCB per poter dare risultato al processo padre tramite wait() → dunque un processo figlio non più attivo nel processo esecuzione ma ancora attivo nel sistema → ovvero ha un suo processo nella tabella dei processi → in quanto deve comunicare il risultato del processo al padre → se dunque il padre non chiede la wait per prendere il risultato il processo rimane bloccato → comunque tiene un po' di memoria per tenere l'informazione, ma non è più attivo sul sistema → il problema dunque si pone nel caso questi processi diventano tanti e quindi occupano troppa memoria!!

Altre funzioni disponibili (per i sistemi unix/linux):

- getpid() – ritorna l'identificatore del processo attualmente in esecuzione
  - getppid() – ritorna l'identificatore del processo padre
  - execv(char\* path, char\*[] args)
  - execve(char\* path, char\*[] args, char\*[] env);
- le ultime 2 costituiscono delle varianti dell'exec! → permettono di fornire il path del programma che sarà trasformato in processo padre → nell'execve vengono anche fornite le *variabili d'ambiente* che sono un ulteriore informazione → queste contengono delle impostazioni che possono essere utilizzate dal processo

### Input/Output dei processi

In UNIX ma anche in Windows i processi eseguiti da riga di comando hanno associato:

- uno stream di **input** (una sequenza indefinita di caratteri) → dal quale il processo legge i dati da processare
- uno stream di **output** → sempre una sequenza indefinita di caratteri dove il processo scrive i risultati
- uno stream di **errori** - dove il processo scrive eventuali errori occorsi durante l'elaborazione
- + I processi si possono comporre connettendo lo stream di output di uno sullo stream di input di un altro processo, usando una *pipe* (tubo) → che è un operatore tipico per questo tipo di operazioni → permette di collegare l'output di un programma nell'input di un altro programma (occhio nel pipe faccio passare solo output)
- + esempio: (concatenazione di 3 processi per fare 1 attività! → comando possibile nelle shell)

- `ls | grep ^a | wc -l`



- l'output di ls lo passo ad un programma detto *grep* → che serve a trovare tutte le righe che iniziano per “a”
- a sua volta l'output del grep viene passato al comando wordcounter che con l'opzione “-l” dice quante righe ci sono nell'output del programma precedente! → in questo caso l'input sono le righe che iniziano con a → il comando grep serve dunque a filtrare l'input iniziale!

Ad esempio:

```
ubuntu@DESKTOP-HSMGJ00:/mnt/c/Users/pierf$ ls | grep ^a
ansel
ubuntu@DESKTOP-HSMGJ00:/mnt/c/Users/pierf$ ls | grep ^nt
ntuser.dat.LOG1
ntuser.dat.LOG2
ntuser.ini
ubuntu@DESKTOP-HSMGJ00:/mnt/c/Users/pierf$ ls | grep ^n | wc -l
3
ubuntu@DESKTOP-HSMGJ00:/mnt/c/Users/pierf$ -
```

→ dunque -l conta appunto le 3 righe che iniziano con “nt”

### Gestione dei processi(su java)

- Java permette di creare dei nuovi processi → questo viene fatto tramite la classe RunTime
- la quale ha un metodo chiamato *exec* con il quale si fornisce il processo da far partire → ritornando un oggetto che rappresenta il processo stesso!

Ad esempio:

```
Process p =
RunTime.getRuntime().exec("notepad.exe");
System.out.println("attendo che finisca...");
p.waitFor(); // attende che il processo generato finisca
System.out.println("finito!");
```

- farò partire il programma notepad.exe sul pc → portandolo in esecuzione → dunque una volta ottenuto il nuovo processo con questo inizia l'esecuzione e dunque ci si mette in attesa che il processo termini! → notare che l'esecuzione della stampa (oppure se aggiungo altre istruzioni) saranno eseguite in concorrenza con il processo che ho eseguito!
- +Ovviamente il file immesso deve essere messo in un posto raggiungibile dal programma!
- notare che RunTime è un singleton → per cui creo un unico oggetto runtime! → e su questo chiamo il metodo exec per far partire il processo che voglio far eseguire! (tornando un process che sarà il processo in esecuzione)

- +In java l'esecuzione diretta di nuovi processi limita la portabilità dell'applicazione → se nell'esempio uso un applicazione del sistema operativo → dunque eseguire notepad.exe funziona sulla macchina windows → ma se portassi il programma su una macchina java con ubuntu ad esempio → avrò un errore! → “notepad.exe” presente solo su windows!
- per fare una cosa del genere risulta quindi preferibile usare i thread → i quali eseguono del codice all'interno della stessa Java Virtual Machine, per cui funziona (anche eseguita in modo concorrente!)

# 31/03/2021

Ripartiamo da un esempio per capire meglio la fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int main()
{
 pid_t pid; /* process id */
 /* fork another process */
 printf("P: sono nel parent\n");
 pid = fork();
 if (pid < 0) { /* error occurred */
 fprintf(stderr, "Fork Failed");
 exit(-1);
 }
 else if (pid == 0) { /* child process */
 printf("F: sono nel figlio\n");
 for(int i=0;i<10;i++) {
 printf("F: %d\n",i);
 sleep(1); //aspetta 1 secondo
 }
 return 5;
 }
 else { /* parent process */
 /* parent will wait for the child to complete */
 int ret = -1;
 for(int i=0;i<5;i++) {
 printf("P: %d\n",i);
 sleep(1);
 }
 printf("P: aspetto figlio %d\n",pid);
 wait(&ret);
 printf ("F: Child Complete %d\n", WEXITSTATUS(ret));
 //sleep(100);
 exit(0);
 }
}
```

→ la macro *WEXITSTATUS* → serve per avere come valore di ritorno unicamente il numero  
indicato → in quanto *ret* contiene più informazioni → dunque la macro serve per vedere il  
risultato di exit del processo figlio (come richiesto nel wait)

```

ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$ gcc fork.c -o fork
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$ ls -l
total 24
-rwxrwxrwx 1 ubuntu ubuntu 123 Mar 26 12:06 cpubound.c
-rwxrwxrwx 1 ubuntu ubuntu 17040 Mar 31 11:14 fork
-rwxrwxrwx 1 ubuntu ubuntu 800 Mar 31 09:58 fork.c
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$./fork
P: sono nel parent
P: 0
F: sono nel figlio
F: 0
F: 1
P: 1
F: 2
P: 2
P: 3
F: 3
P: 4
F: 4
P: aspetto figlio 216
F: 5
F: 6
F: 7
F: 8
F: 9
P: Child Complete 5
ubuntu@DESKTOP-HSMGJOO:/mnt/c/Users/pierf/process$

```

→ notare che eseguo questo programma nella shell linux → in quanto per fare la fork senza altri problemi ho bisogno di un ambiente unix

→ noto quindi che i primi valori che stampo sono **contemporaneamente** del padre e del figlio → finito quindi il ciclo for (del padre → prima del wait) → a quel punto dovrò aspettare che finisca il processo figlio per poi tornare al padre! → il quale tramite la *ret* riesce ad accedere al valore di output del processo figlio! (notare che scrivere return 5 o exit 5 all'interno del processo figlio è la stessa cosa!

+Altro esempio(questo da riprovare) → sulla gestione dei processi in java

```

import java.io.IOException;
import java.lang.Runtime;
public class ProcessTest {

 /**
 * @param args the command line arguments
 */
 public static void main(String[] args) throws IOException, InterruptedException {
 Process p = Runtime.getRuntime().exec("notepad.exe");
 System.out.println("attendo che finisca...");
 p.waitFor(); // attende che il processo generato finisca
 System.out.println("finito!");
 }
}

```

→ in sostanza da windows → il programma apre un nuovo notepad e fino a che questo non viene chiuso il processo continua ad aspettarlo! → solo una volta che esco dal blocco note stamperò "finito!" (notare rispetto all'esempio di prima ho aggiunto la possibilità di lanciare delle eccezioni! → altrimenti il programma NON parte!)

+Dunque continuando la gestione dei processi su Java...

Tramite l'oggetto Process si può accedere agli stream di input/output del processo in esecuzione → tramite l'istanza possiamo eseguire i metodi:

- p.getInputStream() //stream per accedere all'output del processo
  - p.getOutputStream() //stream per fornire input al processo → questo è invece un output per la nostra applicazione tramite la quale possiamo mandare l'input per il nostro processo!
  - p.getErrorStream() //stream per accedere all'stderror del processo
- +Inoltre si può terminare il processo tramite metodo destroy → p.destroy() → il quale termina il processo in esecuzione!

+Esempio di uso del destroy

```
/*
| public static void main(String[] args) throws IOException, InterruptedException {
| try {
| Process p = Runtime.getRuntime().exec("notepad.exe");
| System.out.println("attendo 5 sec...");
| Thread.sleep(5 * 1000); //aspetta 5 secondi
| p.destroy();
| } catch (IOException e) {
| e.printStackTrace();
| } catch (InterruptedException e) {
| e.printStackTrace();
| }
| }
| }
```

→ il programma dunque lancia il processo per il notepad e dopo 5 secondi lo distrugge!

+scrivendo "p." potremmo dare un'occhiata agli altri metodi presenti nella classe... ad esempio children() → per sapere i figli di un processo - descendants() → che opera in modo simile - exitValue() - pid() → che ritornano valori e rappresentano alcune informazioni → allo stesso modo isAlive() (torna un booleano) e info() - infine waitFor() in modo da dare un certo timeout

+Esempio 2:

```
class TestProcess2 {
 static public void main(String[] args) {
 try {
 Process p = Runtime.getRuntime().exec("cmd /c dir");
 InputStream is = p.getInputStream(); //stream per prendere output comando dir
 BufferedReader br = new BufferedReader(new InputStreamReader(is, "Cp850"));
 String line;
 while((line = br.readLine()) != null)
 System.out.println(":> "+line);
 p.waitFor();
 } catch (IOException | InterruptedException ex) { ex.printStackTrace(); }
 }
}
```

Esegue comando dir  
da riga di comando (cmd.exe)

Stampa le righe prodotte  
in uscita dal comando dir

Codifica caratteri  
accentati usata da  
windows

→ in questo programma viene chiesto di aprire la command line → eseguire il comando "dir" che stampa un sacco di belle directory (o almeno quelle sotto un certo utente) → e passo queste righe all' *InputStream* is → che però sarà una sequenza di bit → per cui riconverto tutte in un oggetto *BufferedReader* per poi stampare il tutto su console

→ notare che si è specificato nel bufferedReader “Cp850” in modo da leggere correttamente tutti i caratteri nella specifica di windows! (diversa dalla codifica di UNICODE!)  
→ il metodo getInputStream in realtà mi va a prendere un output dell'applicazione → ma per come funziona viene chiamato inputStream (infatti deve tornare un inputStream!)  
→ notare il blocco catch finale → nel caso volessi gestire più eccezioni contemporaneamente possiamo usare questa sintassi → che fa in un certo senso l'or delle eccezioni! → dunque evita di scrivere diversi blocchi catch → tratto però tutte le eccezioni lanciate alla stessa maniera  
+notare infine che l'output in qualsiasi modo lo si chiama deve essere un input stream → in quanto questi servono a prendere degli input verso l'applicazione! (in questo caso in realtà ottieni l'output di un primo processo → cmd dir)

### Processi cooperativi

Processo **indipendente** → un processo che non viene influenzato e non influenza l'esecuzione di altri processi

Processo **cooperativo** un processo che può influenzare e può essere influenzato da altri processi → l'obiettivo complessivo della realizzazione dell'applicazione è dunque realizzato tramite diversi processi che cooperano per uno stesso obiettivo!

→ abbiamo dei vantaggi di processi cooperativi

- Condivisione di informazioni tra i processi
- Velocizzazione delle elaborazioni → alcuni processi che si occupano di cose diverse lavorano in modo concorrente!
- Modularità → processi diversi si occupano di cose diverse → abbiamo appunto un approccio modulare!
- Convenienza

→ nel caso di processi cooperativi risulta anche più semplice passare ai cosiddetti processi distribuiti! → nel quale le funzionalità dei vari processi le posso mettere su macchine diverse (e non più su un'unica macchina) → da centralizzato a distribuito!

### InterProcess Communication (IPC)

Sono dei meccanismi usati dai processi per comunicare e sincronizzare le loro azioni

→ Ci sono due modelli principali per lo scambio di informazioni tra processi:

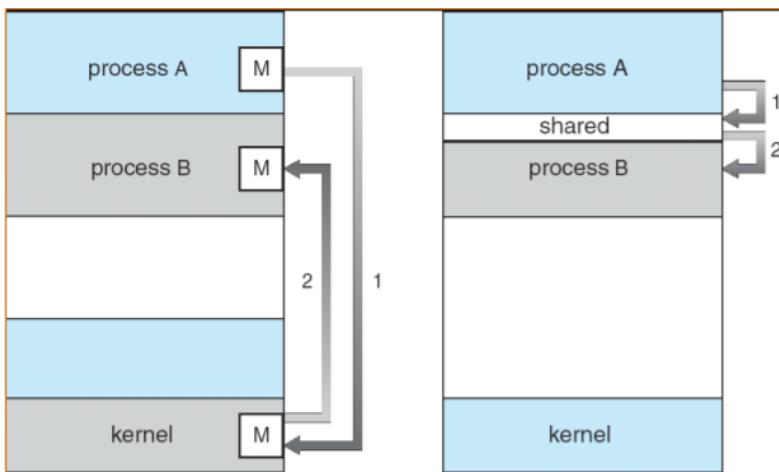
- Memoria condivisa → viene creata una zona di memoria **condivisa** tra due o più processi → più processi usano la stessa memoria
- Scambio di messaggi → dove esistono delle funzionalità del sistema operativo che permettono la funzionalità di scambio dei messaggi!

→ I moderni sistemi operativi spesso forniscono entrambe le modalità. → Infatti la memoria condivisa, più efficiente, viene usata per scambiare grandi quantità di dati

Invece lo Scambio di messaggi, sono meno efficienti, ma utili per sincronizzare processi → usato nel caso uno debba aspettare qualcosa → ad esempio aspettare che arrivi un'informazione da un altro processo → per cui si sincronizza con quel processo in modo che gli mandi la cosa su cui deve lavorare → con la memoria condivisa questo risulta più difficile!

→ infatti tutti i processi possono scrivere/leggere e dunque un processo non sa quando l'altro processo ha letto/scritto il dato che gli aveva scritto → questo comporta qualche difficoltà...

+Esempio modello di comunicazioni



→ a sinistra ho quello basato sui messaggi → processo A che vuole mandare un messaggio al processo B → dunque manda la copia del messaggio al kernel il quale la manda poi al processo → e quindi fare la copia di questi byte se sono pochi avviene in tempo trascurabile → altrimenti per messaggi dell'ordine del megabyte questa copia che dev'essere fatta non diventa più trascurabile!

→ mentre invece nel caso di un approccio con memoria condivisa abbiamo che → il processo A scrive nella memoria e poi il processo B leggerà dalla stessa memoria che è accessibile ad entrambi → per cui il processo A scrive e il processo B legge! (anche se come detto prima rimane il problema che il processo B non sà quando il processo A scrive!  
 → deve quindi rimanere per tanto tempo in attesa e quindi non risulta efficiente!)  
 → per cui riassumendo:

- messaggi per sincronizzare
- memoria condivisa per trasferire i dati!

IPC fornisce due operazioni di base:

- send(messaggio) – messaggio a grandezza fissa o variabile
- receive(messaggio)

Dunque se i processi P e Q vogliono comunicare, devono: prima stabilire un canale di comunicazione tra loro e a quel punto scambiare messaggi tramite le operazioni di send/receive

→ Il canale di comunicazione presenta quindi: un livello fisico (es. memoria condivisa, hardware bus) e un Livello logico (es. proprietà logiche) → per le scambio dei messaggi

## Comunicazione diretta

→ I processi devono nominarsi **esplicitamente**: "ovvero un certo processo Q deve dire che vuole inviare un messaggio ad un determinato processo P" → dunque ho le operazioni di

- send (P, message) – invia messaggio al processo P
- receive(Q, message) – riceve un messaggio dal processo Q

→ Abbiamo dunque in questo caso un canale di comunicazione esplicito che viene stabilito automaticamente!

→ i problemi di questo tipo di comunicazione sono però → primo sapere chi è l'altro (dunque problemi di identificazione) → potremmo ad esempio usare il *pid* → ma dobbiamo stare attenti ad usare il pid giusto... poi il ricevente deve anche lui esplicitare da quale processo

vuole ricevere → se dunque dico che il processo può ricevere da più processi questo non risulta molto agevole!

+Altre caratteristiche sempre di questo tipo di comunicazione sono che un collegamento viene stabilito tra due processi comunicanti (unico modo che hanno per comunicare) → non possono esistere più collegamenti fra i processi per cose diverse  
→ infine il collegamento può essere unidirezionale ma spesso è bidirezionale!

## Comunicazione indiretta

Modello più usato dove abbiamo delle porte/mailbox dove i processi comunicano fra loro  
→ ho dunque delle porte che li metto in comunicazione

→ Una porta ha un id univoco e i processi possono comunicare solo se **condividono** una porta

+Alcune proprietà del collegamento sono:

- Collegamento stabilito solo se i processi condividono una porta in comune
- Un collegamento può essere associato a più processi (relazione molti a molti!)  
→ Ogni coppia di processi può condividere molti collegamenti e come prima il Collegamento può essere unidirezionale o bidirezionale

+Le operazioni possibili sono quindi quelle di poter creare una porta → dunque inviare i messaggi e riceverli tramite sempre una porta e infine poter distruggere una porta

→ le primitive che sono offerte dal sistema operativo saranno:

- send(A, message) – invia un messaggio alla porta A
- receive(A, message) – riceve un messaggio dalla porta A

→ message come nel caso precedente fà da contenitore (dunque una struttura dati) dove poter tenere il messaggio!

→ giustamente il bellini ricorda che questa è un astrazione → ogni sistema operativo avrà le sue chiamate con dei parametri giusti e del tipo giusto a seconda dell'operazione da fare (spoiler sulle *socket* → simili alle porte ma associate all'indirizzo di rete quindi possono essere usate per la comunicazione dei processi all'interno di una rete → anche se lavorano nella modalità appena detta → abbiamo una cosa condivisa nella quale il processo invia dei dati e un modo per riceverli e usare questa cosa condivisa... )

+Abbiamo però il problema di condivisione di una porta → se abbiamo ad esempio 3 processi (P1,P2,P3) che condividono una porta A → se P1 invia un messaggio e sia P2 che P3 ricevono il messaggio → chi se lo prende??

→ Il sistema operativo deve prendere una decisione... e ci sono varie soluzioni:

- Permettere che una porta possa essere associata al più a **due** processi → limitando le possibilità di una porta
- Permettere che solo un processo alla volta possa eseguire la receive su una porta → per cui solo P2 o P3 possono fare la receive uno alla volta → se ad esempio prima lo fa P2 e poi ci prova P3 → questo tentativo viene rifiutato e dunque fallisce di fare la receive
- Permettere al sistema di decidere arbitrariamente a quale processo inviare il dato, al processo inviante viene notificato quale processo ha ricevuto il messaggio.  
→ Ad esempio P1 invia, mentre P2 e P3 sono in attesa di ricevere un dato e allora questo dato verrà preso o da P2 o da P3 → non è indicato esplicitamente chi dei 2 lo riceve, se non il processo che invia il dato riceve in feedback quale processo lo ha ricevuto!

## Sincronizzazione

Queste comunicazioni possono essere sincrone/asincrone, oppure bloccanti/non bloccanti  
→ L'invio/ricezione di messaggi se bloccante è considerato anche sincrono:

- Invio bloccante: il processo viene bloccato fino a che il messaggio non viene ricevuto → se non c'è nessun processo che lo riceve in quel momento → lui aspetta che il messaggio venga ricevuto da un qualche processo alla receive
- Ricezione bloccante: il processo viene bloccato fino a che un messaggio è disponibile!  
→ un processo fà la receive e dunque aspetta che qualcun'altro faccia la send!

→ Invece l'invio/ricezione di messaggi non bloccante è considerato anche asincrono:

- Invio non bloccante: il processo invia il messaggio e continua → il messaggio sarà preso dal sistema e tenuto da parte... se chè il processo che ha fatto la send sà in che stato sia il messaggio → al termine dell'esecuzione della send non si ha la certezza che un altro processo abbia ricevuto il dato inviato!!
- Ricezione non bloccante: il processo riceve un messaggio valido o un messaggio nullo  
→ per dire che non ci sono messaggi! → per cui la receive dà come risultato che o ricevo il messaggio oppure non c'è nessun messaggio!  
→ l'istruzione successiva alla ricezione può essere quindi che ho ricevuto il messaggio, oppure non ho ricevuto il messaggio e quindi ad esempio chiederlo nuovamente aspettando un certo

## Code di messaggi

Per gestire i messaggi il sistema operativo può mettere davanti (nello scambio dei messaggi) delle code → possono quindi esserci code di capacità zero → 0 messaggi per cui il processo inviante deve aspettare il ricevente → nel caso dunque si faccia un invio non c'è posto dove poter mettere questo dato che sto mandando → finchè non arriva il ricevente e quindi lui deve aspettare!

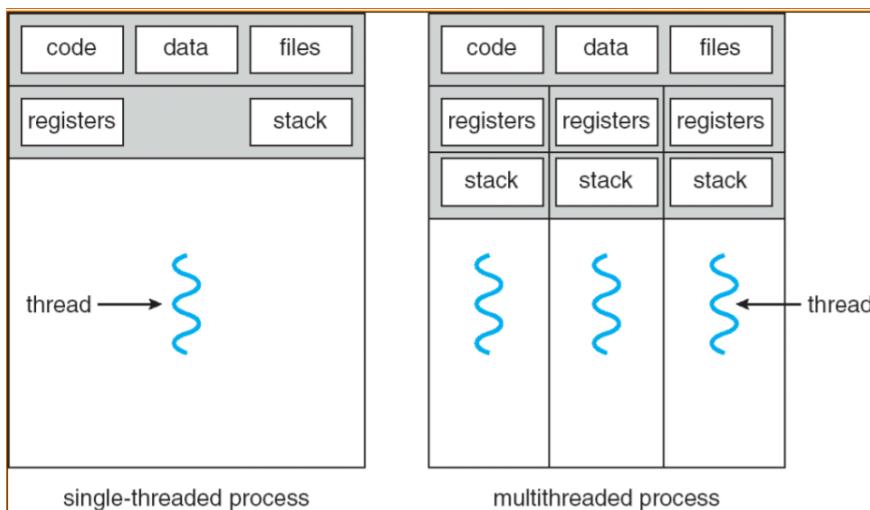
(ripetendo un pò (grazie cristianino) → non c'è un posto dove mettere i messaggi che si stanno scambiando... quindi il processo che invia, deve aspettare che l'altro lo riceva! → glielo mette e finchè l'altro non fà la receive → questo aspetta!) → "rendevous" → i 2 si devono incontrare

+Altrimenti abbiamo capacità limitata → lunghezza finita di n messaggi → in questo caso il processo inviante aspetta solo nel caso la coda sia piena! oppure può anche fallire! → fallisce nel caso di un invio non sincrono!

→ oppure ancora abbiamo capacità illimitata → il sistema ha abbastanza spazio per tenere vari messaggi!

## Threads

I nostri sistemi operativi gestiscono processi single/multi-thread



- nel processo single thread abbiamo un solo / una sola sequenza di istruzioni che vengono eseguite!
- il thread (ovvero filo) rappresenta le istruzioni eseguite dal nostro processo (in questo caso di tipo single-thread) → in alto sono quindi presenti le informazioni usate dal nostro processo → dunque il **codice** → ovvero la sezione di memoria dove sono memorizzate le istruzioni che vengono eseguite
- parte di memoria che contiene i **dati** → che sono in un certo senso le variabili globali della nostra applicazione
- poi abbiamo una parte relativa alla gestione dei **file** → quali file sono aperti oppure in uso da un certo processo!
- abbiamo quindi i **registri** → utilizzati nella computazione del nostro programma e infine abbiamo lo **stack** → nel quale vengono inseriti le variabili locali, oppure punti di ritorno di funzioni, oppure ancora parametri di ritorno ecc...

+Nel caso invece di processo multi-thread abbiamo più sequenze di esecuzioni che vengono eseguite → per cui l'esecuzione poi potrà essere parallela nel caso in cui abbia a disposizione più cpu → e in realtà poi sarà alternata all'esecuzione dei vari thread nel caso avessi un solo core a disposizione! (verrà fatto l'interleave fra questi processi ??)

→ si nota però che ogni thread ha il proprio stack per gestire le istruzioni come variabili locali o punti di ritorno... e ogni thread ha una copia dei propri registri che gli servono per eseguire il codice del thread → i registri dunque non sono condivisibili (allo stesso modo dello stack)

→ avere ad esempio stesse variabili locali fra i vari programmi non sarebbe molto funzionante!

→ Invece frà i vari thread abbiamo condiviso il codice(sezione della memoria dove è presente il codice) → i thread infatti eseguono un certo codice senza modificarlo! → che rimane lo stesso per tutti...

I thread hanno anche variabili globali condivise (per cui anche lo heap) sono condivise fra i vari thread → inoltre anche i file aperti/in uso sono condivisi!

Quali saranno quindi i benefici nell'usare un'applicazione multi-thread rispetto ad una a singolo thread ??

→ Diminuisce i tempi di risposta di una applicazione: ad esempio durante l'attesa di un certo processo, altre attività della applicazione possono essere fatte!

→ concretamente ad esempio mentre in un browser scarico un'immagine da un server potrei fare altre attività... come il rendering (?) o altro... oppure ad esempio nel programma di word mentre stiamo scrivendo qualcosa, il programma controlla cosa abbiamo scritto! (attività che può essere fatta in modo concorrente a quella che stiamo facendo!)

→ Condivisione delle risorse: tutti i thread accedono alla **stessa** memoria e file → se invece dovessimo fare la stessa cosa con più processi... ogni processo avrebbe una propria memoria → per cui ho duplicazioni che portano a spreco di memoria! (e condividono anche l'accesso ai file)

→ Economia: la creazione dei thread costa meno della creazione dei processi! → anche il context switch ha costo minore! → perchè non cambia la zona di memoria usata dai processi! → se ad esempio devo fare il context switch fra i thread dovrò rispristinare solo dei registri! (e non tutto quello che mi permette di accedere all'altra parte della memoria!) → ancora facendo il parallelismo con dei processi diversi → nel caso di context switch tra 2 processi... cambia tutta la zona di memoria che porta via molto tempo! (rileggere cosa era il context switch)

→ Un altro vantaggio è → l'utilizzazione di architettura multi-processore per lo sviluppo di una singola applicazione: possibilità di effettivo parallelismo → ovvero a livello di singolo processo posso usare tutti i core che ci sono a disposizione (per fare effettivamente delle operazioni in parallelo!)

## User/kernel threads

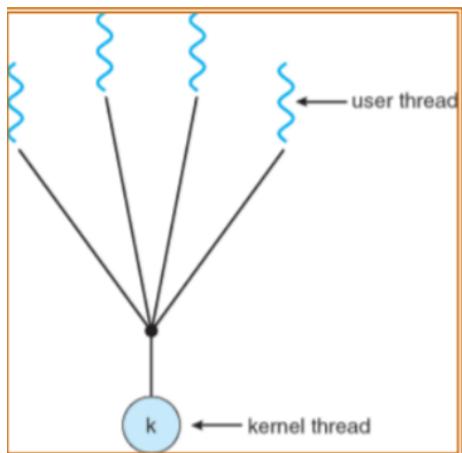
**User Thread:** (thread a livello applicazione) → Gestione dei thread fatto da una **libreria** di gestione dei thread → libreria che si trova negli eseguibili che abbiamo realizzato → per cui risulta gestita a livello utente (no kernel) → in questo caso dunque scheduling è fatto dall'applicazione stessa → applicazione che passa in esecuzione i vari thread che sono in esecuzione

**Kernel Thread:** thread gestiti dal sistema → tutti i sistemi operativi recenti supportano il multi-threading

+ Il legame fra user thread e kernel thread può quindi essere di vari tipi:

- **Molti-a-uno** → molti thread utente gestiti da un solo thread kernel (questo dunque era il modo per implementare il multi-thread quando ancora non era stato integrato nella macchine)
- **Uno-a-uno** → un thread utente gestito da un thread kernel → dunque in questo caso thread utente e quello kernel sono la stessa cosa!
- **Molti-a-molti** → molti thread utente gestiti da molti thread kernel!

## Molti-a-uno



→ vari thread dalle applicazioni che però sono eseguiti da un unico kernel thread → il quale si alterna nell'esecuzione di questi user threads

Quali sono i vantaggi?

→ possono gestire i thread in modo efficiente → in quanto la maggior parte delle operazioni sono fatte a livello utente → per il passaggio dell'esecuzione non entra in gioco il kernel!

→ se però uno di questi thread ha una chiamata di sistema che è bloccante → dunque aspetta che passi l'esecuzione in modalità kernel → tutti gli altri thread non possono essere eseguiti!!

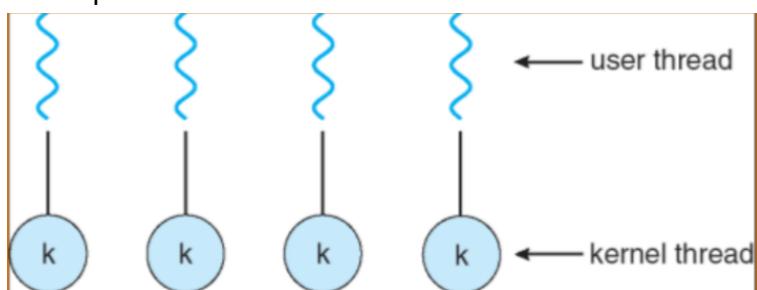
→ un altro svantaggio è il fatto che non usano più core! → le istruzioni vengono tutte eseguite in un solo thread → e dunque non operano in parallelo!

→ infine non è *pre-emptive* → la CPU non viene tolta ad un thread per cui dobbiamo aspettare e anzi dobbiamo far sì che la CPU venga rilasciata esplicitamente → "nell'esecuzione di una thread dirà ora ho finito di usare la CPU, quindi passa ad un altro" e così via passando uno all'altro

→ infine ci sono degli esempi → GNU Portable Threads, Solaris Green Threads

## Uno-a-uno

→ Ogni thread a livello utente gestito da un thread del kernel → per cui creare un thread utente porta alla creazione di un thread del kernel!



→ i vantaggi principali dunque sono il fatto che essendo gestiti dal sistema non si bloccano su chiamate di sistema bloccanti! (se uno fa una chiamata bloccante gli altri thread possono proseguire tranquillamente)

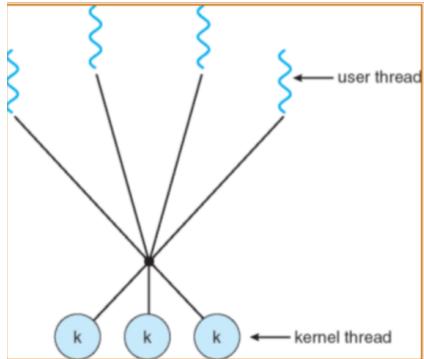
→ e inoltre possono sfruttare il parallelismo dei multi-core del processo

→ lo svantaggio principale però risulta essere che in questo caso il sistema potrà gestire un numero limitato di thread!

→ Windows (da NT in poi), linux e Solaris 9 (e successivi) usano questo tipo di legame fra i thread

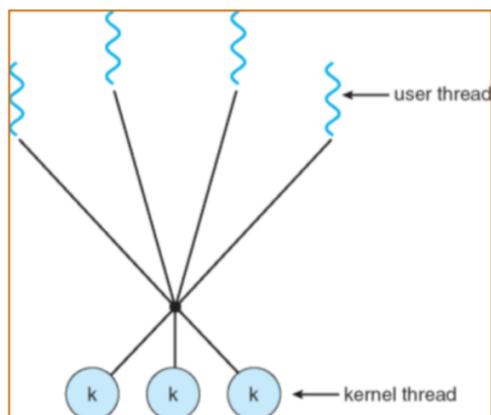
## Molti-a-molti

Si permette a molti thread utente di essere gestiti tramite molti thread kernel



- gli user thread sono molti e generalmente quel del kernel sono in numero inferiore
- nel momento in cui un thread entra in attesa → ovvero fa una chiamata bloccante... gli altri possono proseguire usando gli altri thread del kernel che li possono gestire → l'esecuzione dei thread verrà alternata sull'esecuzione dei thread del kernel! → solo nel caso dei thread eseguono delle chiamate bloccanti, più di quanti sono i thread del kernel, in quel caso ho un problema...
- anche in questo caso si può sfruttare il parallelismo in quanto appunto i thread possono essere eseguiti in parallelo su i thread del kernel → i thread vengono comunque eseguiti a livello utente → per cui ce ne possono essere anche un numero considerevole! (mentre a livello di kernel rimangono comunque ad un numero limitato)
- per cui si hanno gli stessi vantaggi del caso precedente ed in più possiamo aumentare il numero dei thread utente(più di quelli che gestisce un S.O nella possibilità precedente)

+Esiste inoltre un modello ibrido → modello a 2 livelli in cui si ha la possibilità di fare sia l'applicazione molti a molti che uno a uno



→ come esempi c'è windows 7 user mode - scheduling??

## Problematiche programmazione multi-thread

I principali problemi sono:

- Chiamate fork & exec ■ Terminazione dei thread ■ Dati specifici dei thread

## UNIX: Semantica di fork() e exec()

Abbiamo visto la fork e la exec per creare dei nuovi processi figli... ma che succede se facciamo la fork in un thread?? vengono duplicati i thread sul processo chiamante oppure tutti i thread?

→ Ad esempio Linux: duplica solo il thread chiamante → ma può generare errori in quanto gli altri thread possono lasciare memoria del nuovo processo in stato **inconsistente**! → Nel momento in cui viene fatta la fork viene fatta una copia della memoria del processo → e se ci sono dei thread in esecuzione in quel momento → questi ad esempio rimangono a metà! → e questo *rimanere a metà* può generare dei problemi successivamente al processo → se ad esempio si accede ai dati caricati a metà creati dall'altro thread → questo porta a fallimento! → per cui in genera si evita di usare la fork per un'applicazione multithread

## Terminazione dei Thread

Come terminare un thread prima che abbia terminato le operazioni naturalmente??

Abbiamo due approcci principali:

- Terminazione asincrona → termina il thread **immediatamente** (potrebbe non rilasciare le risorse acquisite dal thread → lasciando la memoria in uno stato incosistente) → da evitare! (se ci sono altre possibilità)
- Terminazione ritardata → il thread controlla **periodicamente** se deve essere terminato → si deve fare in modo nella programmazione del thread che lui ogni tanto vada a vedere se deve essere terminato → e in quel caso smetta di fare quello che stava facendo → e inoltre rilascia le risorse, mettendo la memoria in uno stato consistente nel punto in cui termini  
→ di solito è la modalità preferita per la terminazione dei thread  
→ nel caso questa non funzioni (ad esempio il thread risulta bloccato) a quel punto si usa inevitabilmente la terminazione asincrona!

## Dati specifici dei thread

→ i thread condividono una memoria globale che però è associata al thread e non condivisa con gli altri! → è come avere un proprio stack dove salvare le variabili globali, ma che è accessibile solo al proprio thread!

→ alcuni sistemi operativi forniscono quindi la possibilità di creare questo *thread local storage*(TLS) → zona di memoria globale che è associata al thread!

→ per cui tutte le funzioni che vengono chiamate all'interno di quel thread possono accedere a questa zona di memoria condivisa → ma è condivisa solo a livello di thread! → altri thread accedono a variabili globali diverse!

## Linux threads

Linux chiama i threads → tasks → e la creazione dei thread è fatta tramite la chiamata di sistema che è la *clone()* (che in realtà viene usata anche dalla fork() ) → la quale esplicita cosa copiare → se condividere fra padre e figlio determinate cose e nel caso di un thread gli dice di condividere anche la memoria (mentre ciò non avviene per la fork!!) → ognuno si

tiene la sua memoria, mentre nel caso di un thread avranno tutte la stessa memoria (ogniuna avrà invece il proprio stack)

→ clone() permette a un task figlio di condividere lo spazio degli indirizzi del task padre (processo)

+La libreria che si usa su linux/unix per gestire i thread è uno standard POSIX → che permette la creazione e la sincronizzazione dei thread!

→ ci sono delle API standard che sono le stesse su sistemi operativi UNIX diversi ma in qualche modo compatibili

→ La API standard specifica il comportamento della libreria per la gestione dei thread, la implementazione è realizzata da chi sviluppa la libreria → questo dunque risulta comune nei sistemi operativi UNIX come solaris, linux e macOS

→ <http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread.h.html> (link per sbirciare qualche funzionalità)

+Come creare quindi un thread??

→ questo viene associato ad una funzione → che crea un nuovo thread e mette in esecuzione una funzione che gli passo come argomento

```
#include <pthread.h>

...
void* thread_function(void* p){ ... } //funzione da eseguire in thread
...
pthread_t tid; //identificatore del task
pthread_attr_t attr; //attributi per la creazione del thread
...
pthread_attr_init(&attr); //inizializza con attributi di default
void* param = ... //parametro passato a funzione thread
int r = pthread_create(&tid, &attr, thread_function, param);
//ritorna 0 se OK
```

→ la funzione *thread\_function* → deve avere necessariamente come parametro un void pointer → dunque un ptr e lo stesso come valore restituito

→ *pthread\_attr\_t* è in realtà una struttura dati di attributi per cui la funzione successiva (*pthread\_attr\_init*) → serve a inizializzare i dati

→ preso quindi un certo parametro (param) → di tipo ptr a void → si chiama la funzione *pthread\_create* che vuole come parametri il ptr all'identificare del task/thread → poi un ptr ad una struttura dati attr (precedentemente inizializzata) → il nome della funzione che deve essere eseguita sul thread separato e infine il parametro che verrà passato alla funzione come argomento!

+Un'altra funzionalità è quella del thread join che serve a aspettare che termini un certo thread, dato il suo identifier (**task identifier** → tid)

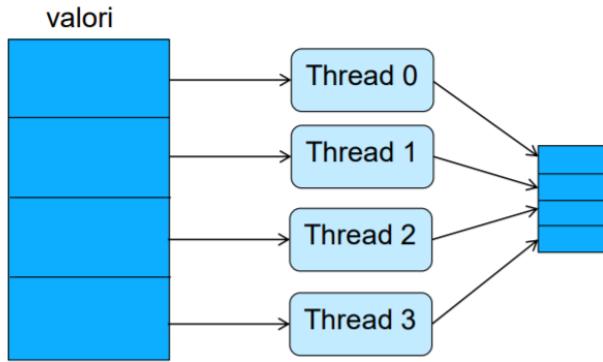
valore ritornato dalla funzione *pthread\_join(tid, NULL);* //aspetta terminazione di tid

void \* ret\_val;

*pthread\_join(tid, &ret\_val);* //aspetta terminazione di tid e mette in ret\_val il valore ritornato dalla funzione

→ passiamo dunque una variabile di tipo void ptr, dove mettere il risultato di una funzione che viene eseguita sul thread (se metto NULL dico che non mi interessa il risultato!)

+Esempio(da saltare): voglio sommare N valori in un vettore usando 4 thread



→ voglio quindi far fare al primo thread la somma dei primi  $N/4$  valori, il secondo i secondi  $N/4$  e così via → ognuno di questi metterà il risultato in un array delle somme → quindi quando tutti i thread hanno scritto il loro risultato → ne facciamo la somma di questi valori ed ottenere il risultato finale

→ nel caso di una macchina multi-core, questi thread verranno eseguiti su core diversi e quindi ognuno andrà ad accedere a dati diversi → per cui potrà eseguire questa attività in parallelo → cosa non possibile con un unico core! → dove il processore si alterna sull'esecuzione dei vari thread (non per forza uno alla volta) → per cui non ottengo un grande vantaggio in termini di tempi di esecuzione!

+Vado quindi a creare un primo programma usando le funzioni viste

```

#include <stdio.h>
#define N 10000
#define NTH 4 //numero thread
int values[N]; //valori da sommare
int sums[NTH]; //somme parziali

void* sum_thread(void* t) {
 int s=0, th, i;
 th=*(int*)t; //numero del thread 0, 1, 2, 3
 for(i=th*N/NTH;i<(th+1)*N/NTH; i++)
 s+=values[i];
 sums[th]=s;
 return NULL;
}

```

→ variabili globali

→ funzione sum\_thread eseguita su variabili locali → le quali saranno appunto allocato sullo stack *privato* del thread

→ ipotizziamo di passare il puntatore ad un intero che contiene quel numero → per sapere quindi con quale thread sto facendo l'operazione → thread 0, thread 1 e così via... in modo che ogni thread vada ad accedere ai valori della propria parte (notare che alla variabile t viene fatto il cast a int ptr → per ottenere il numero del thread → e dopodichè faremo il for delle somme dei valori facendo in modo che il thread 0 prenda la parte da 0 a  $N/4 - 1$ , thread 1 la parte successiva e così via... (si ipotizza che sia divisibile per 4))

→ uscito dal for accede all'array delle somme parziali e nella posizione indicata dall'indice → ovvero il numero del thread → mette il valore di s

+Il programma principale è quindi:

```

int main() {
 pthread_t tids[NTH];
 int i, s = 0, thn[NTH];
 for(i=0;i<N; i++) // inizializza il vettore con 0,1, 2, ... N-1
 values[i] = i;
 for(i=0;i<NTH; i++){
 thn[i] = i;
 pthread_create(&tids[i], NULL, sum_thread, &thn[i]);
 }
 for(i=0;i<NTH; i++){
 pthread_join(tids[i], NULL);
 s += sums[i];
 }
 printf("somma: %d\n", s);
}

```

Non usare **&i**  
Perché?

→ thn[NTH] è un array per tenere gli indici dei threads!  
in posizione 0 metto 0, 1 → 1 e così via...  
NTH <--> numero dei threads

→ per creare i threads si usa appunto il metodo *create* a cui viene passato l'indirizzo dell'array nella posizione relativa appunto al thread che stiamo creando  
→ ad esempio il thread 0 → userà tids[0] e gli viene

passato il proprio indirizzo, si passa quindi un valore nullo, per inizializzare ad un valore di default, il nome della funzione da eseguire nel thread → poi gli viene passato l'indirizzo che sarà l'argomento poi da dare alla funzione di *sum\_thread* in modo da fargli capire quale sia la sezione che deve gestire → thread 0 ottenga 0, thread 1 → 1 e così via...

→ non possiamo usare **&i** (con i che è una variabile locale) → in quanto nel momento in cui si va a chiamare la funzione, dal momento in cui si fa questa chiamata al momento in cui si esegue la funzione → non sappiamo quando avvenga questa... ??

→ non è detto che alla fine dell'esecuzione della funzione di *create* del thread → il thread sia stato già creato ed è entrato in esecuzione... e se ha fatto tutto quello che è necessario per ottenere il valore puntato da *t*! → alla successiva iterazione del programma principale può capire che il valore di *i* sia cambiato ma non il suo indirizzo! → per cui potrò avere che thread diversi accedino alla stessa zona di memoria → che porta quindi a malfunzionamenti!

→ ripetiamo però che il tutto avviene in modo casuale → dipende infatti dai tempi di esecuzione dei vari thread → sia quello principale che quello per la somma! → specialmente nel caso multicore → il passaggio di lettura del valore di *t* → può capitare di avere lo stesso indirizzo per cui si va a leggere lo stesso valore → oppure ancora i primi 2 leggano lo stesso e il 3 un altro!

→ in questo modo qui invece si passa sempre un indirizzo diverso che contiene il valore di *i* (un'altra possibilità sarebbe stata invece di passare per valore invece che indirizzo e dunque trasformare *i*, interpretato come void \* → passando quindi il valore 0,1,2 come se fosse un puntatore!)

+La cosa importante che pierfra vuole evidenziare è che nel momento in cui eseguiamo l'istruzione successiva a quella di *create thread* → non è detto che questo sia già partito - inizializzato e che abbia raggiunto (ad esempio) quella parte di codice che permette di ricavare i valori dei parametri!

→ non sappiamo inoltre in che ordine termineranno i vari threads → per cui dobbiamo aspettare in quanto possiamo prendere i risultati solo quando questi hanno terminato! → ci si mette quindi in attesa della terminazione dei singoli thread → come si fa questo??  
→ si usa la funzione di join → si aspetta quindi che termini il primo thread → quando questo ha finito a quel punto posso accedere all'array e sommare il contenuto a s! → terminato il primo guardo se ha terminato il secondo e faccio la stessa cosa una volta che il thread sia finito (mentre si aspetta per il primo gli altri vanno avanti dunque ci può stare di aspettare meno o addirittura il thread ha già finito) e così via... fino quindi ad avere su s la somma totale che volevo!

## Java Threads

I thread Java sono gestiti dalla Java Virtual Machine ma in realtà sono eseguiti utilizzando i thread del sistema operativo ospite (questo dipende anche dal SO → dove è in esecuzione la jVm)

→ i thread in java possono essere creati in 2 modi:

- Estendendo la classe *thread* (definita in java)
- Implementando un interfaccia chiamata *Runnable*

## Classe thread

Si estende dalla classe thread e dunque si ridefinisce il metodo **run()**, che contiene il codice da eseguire su di un thread separato → una volta creato un oggetto derivato dalla classe thread questo fa partire dalla classe thread che eseguirà il metodo run() → il metodo **start()** invece fa eseguire il run() su di un thread separato! → lo start però potrà essere chiamato una sola volta! → se ad esempio finito il run() di un thread proviamo nuovamente a fare lo start() → verrà lanciata un eccezione! (non eseguibile una seconda volta sullo stesso thread!)

→ dopo è presente il metodo **join()** → che come prima permette di aspettare che il thread termini (in questo caso che il metodo run() finisca e dunque il thread termini la sua esecuzione!) → se il thread viene interrotto con un metodo **interrupt()** → il join() genera una eccezione!

→ esiste infine il metodo statico **sleep()** (che può essere chiamato in qualsiasi punto di esecuzione (è infatti statico) → non solo nei thread) → che serve ad aspettare un certo numero di millisecondi! → anche questo può generare una interruptedException se interrotto +esempio:

```

public class CountDownThread extends Thread {
 public void run() {
 try {
 for(int i=10; i>=0; i--) {
 System.out.println(i);
 Thread.sleep(1000);
 }
 } catch(InterruptedException ex) {}
 }
}
...
CountDownThread cdt = new CountDownThread();
cdt.start(); //parte il thread
cdt.join(); // attende terminazione del thread

```

- notare quindi che nella classe derivata si ridefinisce il metodo **run** che fa il countdown
- si usa il costruttore di default in quanto non ho niente da inizializzare!
- **lo start fa partire il thread** → il quale inizierà dopo un po' (non si sa quando) con l'esecuzione del codice richiesto → per aspettare quindi la fine si richiede il join → potrei aggiungere altri comandi prima del join → il quale mi è utile nel caso dovesse usare un certo risultato che mi deve fornire il thread in modo tale da aspettare prima di andare avanti il risultato richiesto!
- notare che ogni funzione per i thread non ha nessun argomento passato fra i parametri!
- questo perchè ad esempio run() usa gli attributi dell'oggetto per decidere cosa fare !
- quel sleep di 1000 in realtà aspetta solo 1 secondo!
- +Fare una chiamata al metodo run() invece che start(), in questo caso come risultato si ottiene la stessa cosa → se non che in questo caso verrà eseguito nel thread principale e quindi non è un'esecuzione su di un thread separato!(quindi è un errore!)
- il metodo sleep è inserito in un blocco try-catch in quanto può lanciare un eccezione! → per cui va necessariamente gestito il caso in cui si lancia un'eccezione
  
- +è buona norma quando si estende la classe thread, chiamare la classe derivata con qualcosa che contenga il nome thread!

## Interfaccia runnable

- interfaccia runnable che definisce solo un metodo astratto run() → per cui si crea un oggetto della classe thread (diverso da un oggetto di una classe derivata sempre da thread)
- al quale dà come parametro il riferimento all'interfaccia runnable

```

public class CountDown implements Runnable {
 ...
 public void run() { ... }
}

...
CountDown countDown = new CountDown();
Thread t = new Thread(countDown);
t.start(); // fa partire il thread ed esegue countDown.run();
...
t.join(); //attende la terminazione del thread

```

→ il thread dunque andrà ad eseguire (con lo start) il run della classe countdown → come al solito farò un join per aspettare la terminazione del thread

→ notare quindi che si crea una nuova istanza della classe thread (e non una di una sua classe derivata) → fornendogli come argomento l'oggetto (che è un implementazione della runnable) → sul quale chiamare il metodo run → per cui si crea un thread separato e si esegue il metodo run su questo → lo start lo fà quindi partire e porta all'esecuzione del metodo richiesto

→ in questo schema posso anche creare un nuovo thread che esegue il codice sempre dell'oggetto countDown → posso ad esempio creare un nuovo thread che continua il lavoro che era stato fatto da altri!

+Per far ripartire un thread non è possibile usare il metodo start() → il quale genererebbe un'eccezione se eseguito una seconda volta, per cui si deve creare una nuova istanza!

## 16/04/2021

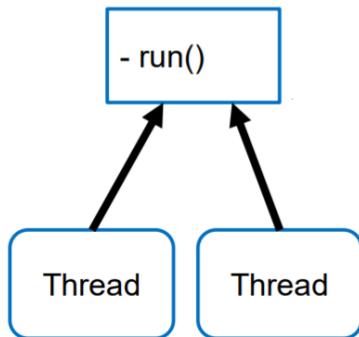
BOING! → saltiamo di 2 settimane per le vacanze di pasqua

Extends o implements

Quando usare l'extend thread oppure l'implement runnable?

→ sicuramente si userà la seconda nel caso la classe sia già in una gerarchia → e dunque non si può estendere anche da thread

→ oppure come detto prima, si usa implement runnable → nel caso si deve eseguire più volte il metodo run() → e preservare lo stato dell'oggetto!



→ per cui eseguito un certo run() all'interno di un thread e si deve mantenere lo stato dell'esecuzione precedente → in modo da continuare un certo lavoro prima eseguito → si crea una nuova istanza di thread che si riferisce allo stesso oggetto e quindi su questo oggetto si continuerà il lavoro che era stato fatto precedentemente (usando l'extends dovremmo creare un nuovo oggetto → reinizializzarlo ecc... dunque più laborioso)

## Terminare un thread (parte 2)

→ esiste metodo **stop()** per terminare un thread in modo asincrono → ma è deprecato per i problemi che può potenzialmente introdurre → il thread viene infatti terminato qualsiasi cosa stava facendo e quindi se erano state acquisite delle risorse queste non sono rilasciate!

→ la soluzione più sicura è usare metodo **interrupt()** che non termina il thread ma indica che il thread deve essere terminato → lo mette in uno stato che gli dice che deve essere interrotto

→ Se però il thread è in attesa con sleep, join etc., si interrompe questa attesa e viene generata l'eccezione **InterruptedException** → che indica appunto che il thread deve essere interrotto! → nell'esempio del programma precedente (il countdown) → se durante la sleep arriva un segnale di interrupt → allora questa lancia l'eccezione che sarà quindi gestita facendo direttamente terminare il programma! (nel caso avessi gestito l'eccezione in maniera diversa... ad esempio localmente sulla sleep → allora avrei gestito il catch solo sulla sleep ed avrei continuato a fare il for con l'interazione successiva → dunque avrei interrotto solo l'attesa di un secondo ed avrei continuato il thread!)

→ Se invece il thread non entra mai in attesa → allora l'interrupt() precedente non funziona!  
→ il programma infatti continua fino alla fine

→ invece può essere controllato regolarmente lo stato tramite metodo **Thread.interrupted()**

→ per capire se il thread è stato interrotto → ovvero è stato richiesto da un altro thread che venga interrotto! → dunque ripetutamente si deve controllare se è stato interrotto o meno!  
Di solito la soluzione migliore è lanciare eccezione **InterruptedException** → in modo da terminare l'eccezione e terminare!

```

void run() {
 while(...){
 ...
 if(Thread.interrupted())
 throw new InterruptedException();
 }

```

→ ovviamente possiamo mettere la funzione di controllo dell'interruzioni anche in altri punti  
→ ad esempio in metodi chiamati indirettamente da questo codice qua! → oppure ancora

nel caso il controllo viene fatto di rado nel codice (dunque non in un ciclo while) → se ci accorgiamo che quando ha richiesto di interrompere il thread, la cosa più semplice è guardare se la condizione è avvenuta e dunque lanciare l'eccezione!

+Esempio di terminazione:

```
CountDownThread cdt = new CountDownThread();
```

```
cdt.start();
```

```
Thread.sleep(3500);
```

```
cdt.interrupt();
```

→ durante i 3 secondi e mezzo viene fatto il countdown a quel punto alla stampa del quarto

→ viene fatto l'interrupt! → si vedrà quindi che il countdown thread fa le prime 3 stampe e poi sarà interrotto!

## Altri metodi per la gestione dei thread in java

+ il metodo **boolean isAlive()** indica se il thread è vivo o meno → ovvero se il thread è già terminato, oppure è ancora in esecuzione

+ prima di chiamare start si può impostare se il thread è un *daemon* o no (per default non lo è) tramite metodo **setDaemon(boolean)** → la caratteristica dei demon è che se il programma principale arriva al suo termine → normalmente la jvm aspetta tutti quei thread che non sono dei daemon vengono terminati → finché loro non finiscono la JVM aspetta → viceversa nel caso di thread impostati a daemon, si arrivi alla terminazione del programma principale in questo caso la jvm non gli aspetta!

→ riassumendo La JVM termina la sua esecuzione solo quando sono terminati tutti i thread che non sono daemon

+ il metodo **yield()** serve a rilasciare la CPU per eseguire il prossimo thread ready → dunque concede la cpu al seguente thread senza aspettare che questo finisca! → se non c'è nessun thread pronto allora continua a funzionare!

+ Si può impostare la **priorità** del thread con un valore tra 1 e 10 (1 minimo, 10 massimo e 5 default) → La politica di esecuzione è che vengono eseguiti prima tutti i thread a più alta priorità → quando la coda dei thread pronti a più alta priorità è vuota passa alla coda a priorità subito inferiore etc. (ma questo dipende dalla implementazione della JVM)

+ Infine di solito l'esecuzione dei thread è time-sliced → pierfra dice che lo rivedremo con la gestione della cpu (ma anche qui dipende da implementazione della JVM)

## Stato dei thread

Ogni thread ha uno stato che può essere:

- NEW, thread creato ma non partito
- RUNNABLE, thread in esecuzione o pronto
- BLOCKED, thread bloccato in attesa lock accesso monitor (anche qui vedi dopo)
- WAITING, thread in attesa indefinita
- TIMED\_WAITING, thread in attesa *temporizzata* (tipo una sleep)
- TERMINATED, thread terminato

Si possono avere quindi delle transizioni:

- NEW → RUNNABLE, oppure da runnable
- RUNNABLE ↔ BLOCKED • RUNNABLE ↔ WAITING
- RUNNABLE ↔ TIMED\_WAITING (dunque possono fare avanti e indietro) e infine
- RUNNABLE → TERMINATED

+Partono ora una serie di esercizi per è bene rivederli quando preparerò l'esame  
→ notare che anche la join() può generare un interrupt exception! → dunque va gestita!

```
class CountDownLatch extends Thread {

 public void run() {
 for (int i = 10; i >= 0; i--) {
 System.out.println("count " + i);
 try {
 sleep(1000);
 } catch (InterruptedException e) {
 System.out.println("interrotto");
 }
 }
 }
}
```

→ come detto prima → questo è il modo di lanciare l'eccezione internamente al programma  
→ per cui il risultato sarà che ad un certo punto si lancia l'eccezione e sarà gestita  
stampando il messaggio, ma successivamente il programma riparte arrivando fino al count  
di zero!! → terminando normalmente come fine del programma!

+Con setName o getName rispettivamente si può settare/ottenere il nome del thread (sono  
metodi della classe thread credo → penso quindi ridefinibili)

```
/*
public static void main(String[] args) throws InterruptedException {
 CountDownLatch[] cc=new CountDownLatch[3];
 for(int i=0;i<cc.length; i++) {
 cc[i]=new CountDownLatch(10,1000);
 cc[i].setName("counter-"+i);
 cc[i].start();
 }
 System.out.println("aspetto ...");
 for(int i=0;i<cc.length; i++) {
 cc[i].join();
 }
}
```

→ in questo caso creo un array di countdownThread (inizializzati con un certo costruttore) →  
ma la cosa interessante è quando stampo l'output su console → noto che i vari thread  
operano in modo parallelo → ovvero non faccio il countdown del thread-0 aspettando di  
andare da 10 a 0 e quindi passando al successivo thread → ma faccio il  
contemporaneamente dei vari thread (in questo caso 3) e in ordine abbastanza casuale!

```

counter-1: count 10
counter-0: count 10
counter-1: count 9
counter-0: count 9
counter-2: count 9
counter-1: count 8
counter-2: count 8

```

→ l'ordine di esecuzione non risulta quindi predicibile!

+Cambiando il tempo di attesa dei vari thread a seconda del loro indice:

```
cc[i]=new CountDownLatch(10,1000*(i+1));
```

→ i thread continuano a lavorare in contemporanea se non che alcuni thread possono finire prima degli altri! (in questo caso ad esempio l'output su console finirà con il countdown di del counter-2

+

```

 System.out.println("aspetto 10sec...");
 Thread.sleep(10000);
 for(int i=0;i<cc.length; i++) {
 //cc[i].join();
 cc[i].interrupt();
 }
 //c.join();
 //Thread.sleep(3500);
 //c.interrupt();

```

---

```

-thread-count (run) x
counter-2: count 7
counter-0: count 1
counter-2: interrotto
counter-1: interrotto
counter-0: interrotto
BUILD SUCCESSFUL (total time: 10 seconds)

```

→ questo invece se aspetto dieci secondi e poi chiamo la interrupt! → per cui passati 10 secondi fermo tutti e 3 i thread in contemporanea

+esercizi(da rifare)

1.Implementare la somma multi-thread di un vettore di interi

→ notare che non ci sono metodi di interruzioni... per cui se volessi terminare/interrompere il thread dovrei usare il metodo *interrupted* → per capire se il metodo era stato interrotto!

```

public void run() {
 System.out.println(getName()+" start");
 sum = 0;
 for(int i=start; !Thread.interrupted() && i<start+n; i++) {
 sum += data[i];
 }
 System.out.println(getName()+" sum="+sum);
}

```

→ se dunque viene chiamato il metodo interrupt su di un thread che stava facendo le somme questo termina l'esecuzione!

+Oppure in modo un po' più complicato...

```

public void run() {
 System.out.println(getName() + ": start");
 sum = 0;
 try {
 for (int i = start; i < start + n; i++) {
 sum += data[i];
 if (Thread.interrupted()) {
 throw new InterruptedException();
 }
 }
 System.out.println(getName() + ": sum=" + sum);
 } catch (InterruptedException e) {
 System.out.println("interrotto");
 }
}

```

→ lancio infatti una possibile interruzione che dunque dovrei gestire!

→ ricordo che faccio questo in quanto non è presente ad esempio un metodo sleep() → o qualcosa che mi faccia entrare in attesa questo non è necessario!

+ notare quindi l'uso dei metodi start() → che fa partire il thread portando ad esecuzione il metodo run() → e il metodo join() → che mi serve ad aspettare che tutti i thread abbiano calcolato la somma dei valori nel loro range → per cui ne posso fare la somma totale finale (occhio al fatto che la join può generare un InterruptedException → va gestita!)

+notare infine che nel caso la dimensione dell'array dei dati non sia divisibile per il numero di thread → allora alcuni valori finali potrebbero avanzare! → dovremmo dunque in qualche modo riuscire ad individuare questi valori e sommarli a quelli riservati all'ultimo thread! → “l'ultimo non gli devo passare solo n, ma n + qualcosa” → come ottenere il qualcosa?

2.Ricerca multi-thread di un valore in un vettore di interi(senza che i valori siano ordinati) → dovrei quindi fare una ricerca sequenziale... per cui si velocizza il tutto usando i thread!  
 → si nell'esercizio 1. che in questo il vantaggio sta nel fatto di usare più core! (somme fatte in parallelo su più core → anche se non un numero esagerato di core... altrimenti ci metto più tempo → pari al numero di core) → sono inoltre entrambi dei programma *cpu-bound* → usano la cpu senza fare mai l'I/O

+Un modo per inizializzare un vettore con valori casuali è il seguente

```

public static void main(String[] args) throws
 int[] data = new int[10000];
 for (int i = 0; i < data.length; i++) {
 data[i] = (int)(Math.random()*100);
 }
}

```

→ la funzione math.random() → ritorna un valore qualsiasi compreso fra 0 e 1 (1 escluso) → in double → per cui una volta che moltiplico per 100 ne faccio il cast a int per avere un array di interi

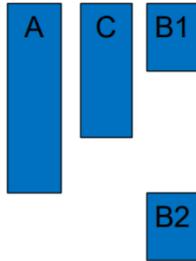
→ notare quindi se un numero decimale ha 2 cifre dopo il punto pari a zero allora sarà comunque interpretato come uno zero in valore intero!

+Per come fornisce la soluzione il bellini, può capitare che ogni thread trovi nella sua parte di array rispettiva di trovare il valore cercato (anzi capita sempre → per la dimensione dell'array e per i pochi valori generati → da 0 a 99) → per cui potrei pensare di far terminare la ricerca appena un thread trovi il valore cercato → e dunque faccia terminare gli altri... come fare?? → christianino dice di usare un interrupt!

3.Prese 3 attività A, B, C dove B risulta divisa in 2 fasi B1 e B2 → voglio fare in modo che B2 usi i risultati prodotti da A,C e B1 → dunque questa potrà essere eseguita soltanto quando gli altri hanno terminato la loro esecuzione

→ userò quindi 3 thread → uno per ogni attività, ma l'attività B dovrà in un certo modo eseguire la parte B1 e finita questa si deve mettere in attesa del risultato di A e di C → per cui il thread dovrà avere conoscenza di queste 2 attività e usare i dati prodotti da A e da C solo dopo che questi siano terminati! → andrà quindi a fare una join per aspettare la fine di A e C e al quel punto eseguo B2

→ notare che nel caso una run() abbia al suo interno il metodo sleep() → non possiamo semplicemente aggiungere throws InterruptedException → in quanto ne stiamo facendo l'override dalla classe base Thread → per cui si deve mantenere la stessa signature!  
→ l'unico modo per gestire l'eccezione è quindi scrivere un blocco try-catch!

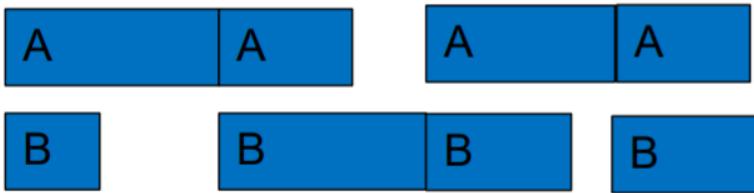


Si fa quindi in modo che l'attività A duri più di tutte (impostando il metodo sleep() ad esempio a 6 secondi), poi C un po' meno e B meno di tutte → dunque una volta finito B1, aspetto le attività A e C (facendo i join() ) e a quel punto inizio B2

→ nel main quindi passiamo all'attività B, i thread A e C, in modo che ne possa invocare il metodo join → quando abbiamo quindi queste necessità risulta necessario fare riferimenti a thread diversi!

→ ricordo inoltre che creare e basta i thread non serve a nulla! → infatti per metterli in esecuzione dovrò usare il metodo start!

5.Siano A e B due attività periodiche → con il vincolo che quando A o B termina → questa verrà avviata nuovamente solo se anche l'altra è terminata → per cui A e B ripartono insieme → solo quando sono finite entrambe → Ad esempio:



→ per cui ipotizzando che B duri meno di A, B ripartirà solo quando termina anche A! → ma può capitare anche il contrario → ma comunque devono ripartire insieme!

+Supponiamo inoltre che le attività portino avanti uno stato → per cui non vogliamo che tutte lo stato venga riazzzerato ma questo continui ad esistere → devo quindi usare l'interafaccia runnable!

+Occhio al fatto che se voglio usare il metodo sleep in una classe che implementa runnable → devo specificare esplicitamente di usare thread.sleep!

→ come gestisco quindi i risultati dei 2 thread?? → devo creare una classe derivata da thread che riceve dei riferimenti ai due thread A e B che ho creato precedentemente → in modo posso decidere quando far partire i 2 thread → usando start e successivamente usato join dei 2 thread in modo da aspettare la fine di entrambi!

```
public static void main(String[] args) {
 ActivityA at = new ActivityA();
 ActivityB bt = new ActivityB();

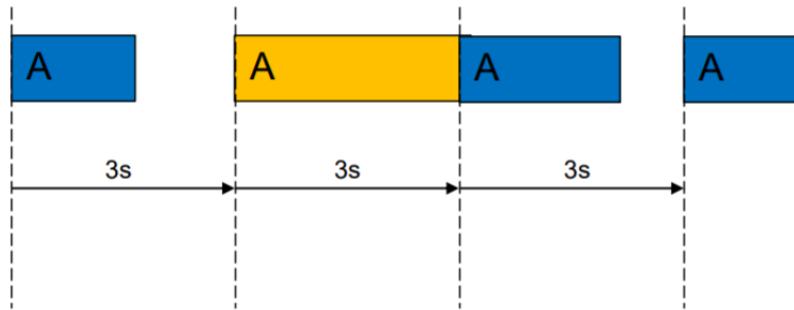
 CycleThreadAB cab = new CycleThreadAB(at,bt);
 cab.start();
 Thread.sleep(30000);
 cab.interrupt();
```

→ come interrompere un programma?? usando interrupt()! → dopo 30 secondi faccio partire il metodo che interrompe il programma! → sfrutto il fatto che siano presenti delle join! → per cui nel mentre che aspetto interrompo il programma (lanciando un'interruzione) → infatti non si ripartirà a fare lo start dei nuovi thread ma si esce dal programma! → notare che comunque interrompo il programma le attività che erano rimaste sospese finiranno!! (per cui ci si chiede come interrompere anche i thread che sono in esecuzione?)

## 21/04/2021

Continuando con gli esercizi sui thread...

6. Ho un'attività A che normalmente dura intorno a 2 secondi ma nel 10% dei casi può superare i 3 secondi... che risulta essere la frequenza di attivazione del thread → si vuole avere quindi il vincolo nel quale nel momento in cui deve essere avviata una nuova attività è ancora attiva la precedente, quella va interrotta e va fatta partire quella nuova!



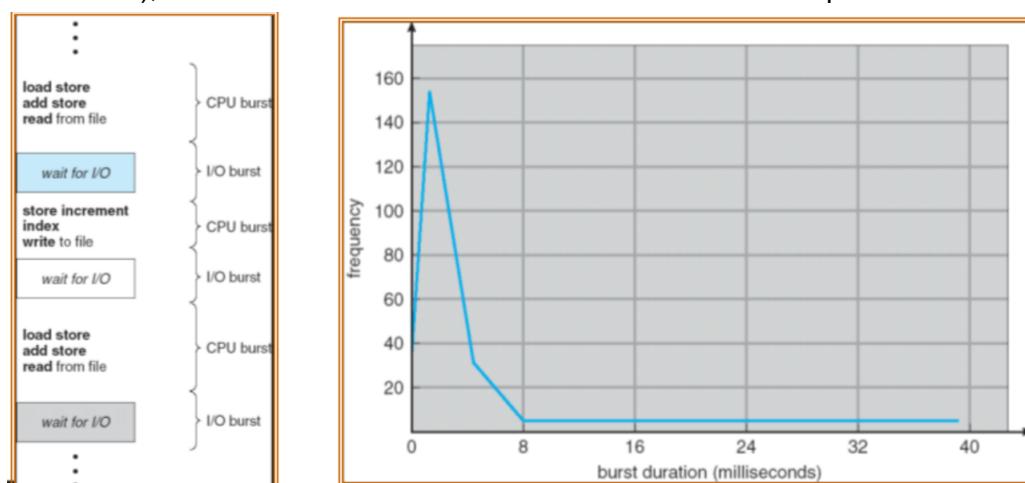
- in modo da non aver quindi attive due attività in parallelo! (un'altra *politica* poteva essere quella di aspettare e non far partire finché l'attività non ha terminato!)
- Come per il resto degli esercizi anche questo è da rivedere!
- Per capire se il thread precedente è ancora attivo o meno → si usa il metodo `isAlive()`! → per cui se da come risultato lo interrompo forzatamente → con `interrupt()`

+Notare poi nel programma posso sostituire il tipo del parametro passato come riferimento alla classe derivata `CycleTimeoutThread` → che al posto della `ActivityA` (che implementa la classe `Runnable`) → posso direttamente usare la classe `Runnable`! → per cui potro estendere a qualsiasi classe implementa l'interfaccia `Runnable` (o in generale derivate da `Runnable`) e il programma continua a funzionare!

## Scheduling della CPU

Ci si domanda ora come i sistemi operativi vanno ad assegnare la CPU (nel caso particolare di singola CPU) ai vari processi che devono essere eseguiti?

- Per usare al massimo la CPU si usa la pratica della *multiprogrammazione* → ovvero sfruttando le fasi in cui un processo è in attesa, per assegnare ad un altro processo la CPU che prima era stata utilizzata
- i processi dunque non fanno altro che passare da una fase in cui usano la CPU (detta CPU-burst), ad una fase in cui sono in attesa di I/O ovvero nel quale non usano la CPU!



- i processi dunque alternano queste due fasi fino al termine del programma → ogni processo ha una sua distribuzione sulla durata dei CPU burst → quanta CPU occupa

durante le varie esecuzioni il processo → per cui avremmo tanti CPU burst che durano poco → dunque a breve durata (ma ad alta frequenza) e pochi che durano tanto!  
→ dal diagramma dunque si possono vedere che sono presenti dei processi I/O bound → che usano poca CPU → dunque hanno quel picco molto alto → e invece ci sono delle applicazioni CPU-bound (con la *pancia* più larga) → che dunque tendono ad usare più CPU → per cui avere una certa frequenza anche per burst più lunghi  
→ dunque a seconda della prevalenza di fasi I/O burst o CPU burst avremmo una determinata distribuzione per ogni diverso processo!

### CPU Scheduler

Seleziona tra i processi in memoria che sono pronti per essere eseguiti e alloca la CPU a uno di questi (scheduling di *breve termine*) → si ipotizza di avere una sola CPU  
→ Le decisioni di scheduling della CPU su un processo possono avvenire quando questo passa:

1. **da in esecuzione in attesa** → il processo in esecuzione ad un certo punto entra in attesa → dunque la CPU viene liberata e il sistema operativo dovrà decidere quale altro processo mettere in esecuzione (primo caso in cui il sistema operativo deve compiere una decisione)
2. **da in esecuzione a pronto** → transizione dallo stato di esecuzione a pronto → caso nel quale la CPU viene tolta dall'esecuzione di un processo per passarla ad un altro! → altro caso in cui il sistema operativo sceglie quale processo mettere in esecuzione
3. **da in attesa a pronto** → un processo nello stato di attesa ad esempio riceve i dati di cui aveva bisogno e quindi passa allo stato pronto all'esecuzione → anche in questo caso il sistema operativo deve decidere quale processo mettere in esecuzione
4. **terminazione** → processo termina quindi libera la cpu che è pronta ad ospitare un nuovo processo

+Lo scheduling che avviene solo nei casi 1 e 4 è detto *nonpreemptive* (senza prelazione) o *cooperativo* → nei casi 1 e 4 infatti il processo abbandona volontariamente la CPU → perchè o si mette in attesa di qualcosa oppure è terminato → con cooperativo si intende il fatto che il sistema operativo coopera con i processi per decidere quale mettere in esecuzione!

→ Se anche in casi 2 e 3 lo scheduling e' *preemptive* (con prelazione) → in questi casi infatti il processo in esecuzione attualmente può essere tolto e al suo posto se ne mette un altro!  
→ può quindi essere interrotto qualcosa che è in esecuzione! → nel caso 2 si vede bene, mentre nel caso 3 ad esempio nel caso di un processo a più alta priorità → il quale deve essere eseguito prima degli altri e quindi si deciderà di far passare lui rispetto ad altri che sono presenti in coda  
→ nei casi di sistemi embedded sono presenti dei scheduling cooperativi → in quanto eseguono le operazioni minime per un sistema operativo, ma in generale i sistemi operativi moderni sono preemptive

### Scheduling preemptive

- Lo scheduling con prelazione può portare a problemi di **inconsistenza** dei dati condivisi tra due processi/thread (*race conditions*) → se condividono dei dati nel caso la CPU viene tolta durante ad esempio un aggiornamento dei dati → questo può portare ad una inconsistenza dei dati!

- Il caso tipico è quello di un thread che viene interrotto durante l'incremento di una variabile in memoria
- Si devono quindi usare meccanismi di **sincronizzazione!** (per appunto evitare problemi di incosistenza dei dati)
- + Ma quando il processo è sottoposto a prelazione, il codice in esecuzione può essere quello del kernel (il processo in quel caso potrebbe essere in una chiamata di sistema) → quindi anche lo stesso kernel deve proteggere le proprie strutture dati (es. code attesa devices) da un uso concorrente → e quindi problemi di inconsistenza (sempre nel caso di prelazione)
- In alcuni sistemi operativi questo problema viene risolto facendo in modo di attendere il completamento della chiamata di sistema prima del cambio contesto → dunque non si puo interrompere un processo durante una chiamata di sistema!!

## Dispatcher

- Il modulo Dispatcher del sistema operativo dà il controllo della CPU al processo selezionato dallo scheduler; → si occupa di:
- Cambiare contesto → ovvero fare il cosiddetto *context switch* → dunque riaggiornare lo stato della CPU (salvando prima quello attuale)
  - Passare a modalità utente
  - Saltare nella locazione opportuna nel programma utente per riprendere l'esecuzione

+Si parla quindi di Latenza di dispatch → tempo impiegato dal dispatcher per fermare un processo e far partire un altro → latenza che si richiede essere la più piccola possibile → in quanto è solo tempo perso dalla CPU per alternare i processi! (se fatta ad esempio durante l'attesa dell'I/O → che può essere molto più lunga rispetto al dispatch ok, mentre nel caso mi serve solo per alternare l'esecuzione di più processi nell'accesso alla CPU in quel caso la latenza di dispatch può essere pesante se si fanno alternare troppo di frequente questi processi!!)

## Criteri di scheduling della CPU

L'obiettivo rimane quello di utilizzare al massimo la CPU → ovvero tenere la CPU più occupata possibile! → i criteri che si usano sono quindi:

- Massimizzare la **Produttività** → numero di processi completati nell'unità di tempo (detta anche *throughput* del sistema) → produrre più risultati completando nel minor tempo possibile i vari processi
- **Tempo di completamento o tempo di ritorno** → tempo necessario ad eseguire un processo specifico → dalla sua attivazione al momento della sua terminazione → detto quindi *turnaround time*
- **Tempo di attesa** → tempo speso dal processo nella coda di attesa dei processi pronti → ripetiamo che il tempo che il processo sta in attesa è un tempo perso dal processo → infatti questo ha tutto pronto all'esecuzione se non che non ha lo spazio necessario all'esecuzione sulla CPU! (e quindi deve aspettare)
- **Tempo di risposta** → tempo impiegato da quando un processo è *sottomesso* a quando viene eseguita la sua prima istruzione → dunque è il tempo impiegato per cui un processo inizi la sua elaborazione!

→ sistemi operativi con tempi di risposta limitati saranno quelli che prediligono la parte interattiva → dove si richiede un feedback dell'applicazione, molto veloce

+per cui Riassumendo i criteri di ottimizzazione sono:

- Massima utilizzazione della CPU
- Massima produttività
- Minimo tempo di completamento
- **Minimo tempo di attesa**
- Minimo tempo di risposta

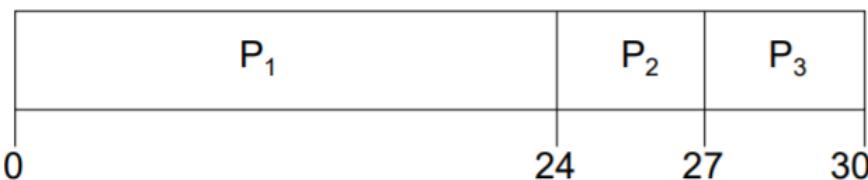
→ Si ottimizzano in realtà o i valori medi oppure i valori minimi/massimi a seconda dei vari processi → Lo scheduling impatta principalmente sul tempo di attesa → per cui questo sarà l'obiettivo principale che in realtà include anche gli altri criteri... ad esempio per la produttività → minore è il tempo di attesa e maggiore sarà la produttività!!

Scheduling First-Come, First-Served (FCFS)

→ Tipo di scheduling più banale → ovvero chi prima arriva meglio è servito!

| <u>Processo</u> | <u>T. esecuz.</u> | <u>T.arrivo</u> |
|-----------------|-------------------|-----------------|
| $P_1$           | 24                | 0               |
| $P_2$           | 3                 | 0               |
| $P_3$           | 3                 | 0               |

→ i tre processi arrivano tutti nello stesso tempo e sono quindi tutti nella coda dei processi pronti → e sappiamo inoltre quanto tempo di CPU userà questo processo → e un processo senza prelazione e come abbiamo detto prima si suppone che i processi siano presenti in coda nell'ordine  $P_1, P_2, P_3$  → per cui saranno serviti esattamente nell'ordine d'arrivo!



→ la seguente figura è detta *diagramma di gant* della schedula → dove si nota che prima viene eseguita tutta  $P_1$ , poi  $P_2$  e infine  $P_3$  → ne possiamo quindi calcolare:

- Tempo di attesa per  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$  (per ogni singolo processo) →  $P_3$  ha dovuto aspettare la fine sia di  $P_1$  che  $P_2$ !
- Tempo **medio** di attesa:  $(0 + 24 + 27)/3 = 17\text{ms}$
- Tempo **medio** di completamento:  $(24 + 27 + 30)/3 = 27$  → ne calcolo il tempo da quanto entra a quando esce!
- Tempo **medio** di risposta:  $(0 + 24 + 27)/3 = 17$  → tempo che passa dall'arrivo alla prima esecuzione della sua prima istruzione

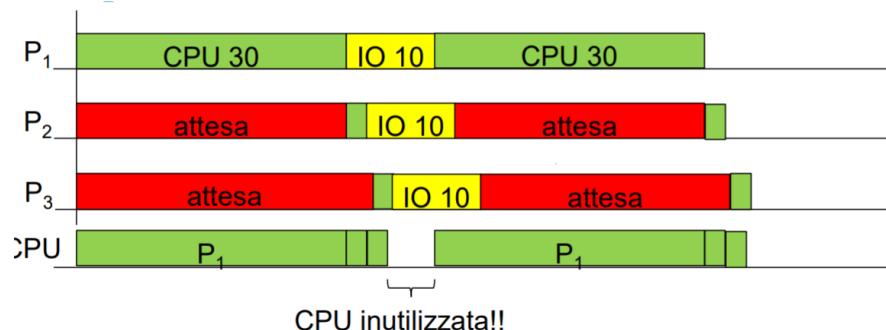
+Cosa succede se invece gli stessi processi arrivano in un diverso ordine? →  $P_2, P_3, P_1$

| P <sub>2</sub> | P <sub>3</sub> | P <sub>1</sub> |
|----------------|----------------|----------------|
| 0              | 3              | 6              |

- Per cui i tempi di attesa saranno sicuramente diversi!
- Tempo di attesa per P2 = 0; P3 = 3; P1 = 6; → e calcolando il tempo **medio** di attesa avrò  $(6+0+3)/3 = 3$  → molto meno rispetto a 17! → per cui risulta più conveniente per il tempo medio di attesa eseguire prima processi più corti!
- discorsi simile per il Tempo medio di completamento:  $(3 + 6 + 30)/3 = 13$  e il Tempo medio di risposta:  $(0 + 3 + 6)/3 = 3$
- la schedula risulta quindi migliore della precedente! → nel quale in particolare avviene l'effetto convoglio → per processi corti dietro a processi lunghi  
(notare che per la politica FCFS → tempo medio di attesa e risposta sono gli stessi!)

+Vediamo ancora un altro esempio:

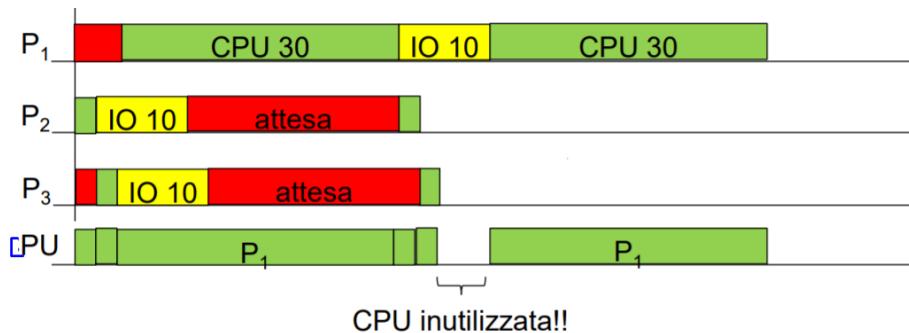
- Tre processi:
- P1 : CPU(30) IO(10) CPU(30) arrivo: 0
  - P2 : CPU(2) IO(10) CPU(2) arrivo: 0 ■ P3 : CPU(2) IO(10) CPU(2) arrivo: 0
  - dove con CPU si intende il tempo di utilizzo della CPU → e lo stesso per I/O, ma imput/output
  - per cui il diagramma di gant risulta → che ci permettono di osservare come la cpu viene assegnata ai vari processi



→ Il Processo P1, dunque arriva per prima e dunque usa la CPU fino a che passa allo stato di I/O → a quel punto lascia la CPU ed entra in esecuzione (seguendo l'ordine di arrivo) P2 → il quale era stato in attesa fino a quel momento → questo processo dunque esegue per 2 ms e appena passa allo stato di I/O allora si passa alla esecuzione del processo successivo → ovvero P3! → non essendoci quindi nessun altro processo pronto all'esecuzione → la CPU rimane per un periodo inutilizzata → terminato quindi l'I/O del primo processo → sarà di nuovo lui a rientrare in esecuzione in quanto l'unico presente nella coda! → per cui esegue fino a terminare l'esecuzione → nel mentre gli altri 2 processi finiscono il periodo di I/O per cui entrano nella coda ready e rimangono in attesa per tutta l'esecuzione del processo P1 → finito questo anche gli altri 2 finiscono la loro esecuzione in sequenza

- In questo caso quindi il tempo di attesa **medio** sarà dato da  $(0+58+60)/3 = 39.33\text{ms}$
- contando infatti solo il tempo di attesa per il processo P2 avro 30+28, mentre P3 32+28

+Cosa accade invece se con gli stessi processi ma l'ordine di esecuzione è diverso? → P2,P3,P1



→ notare che inizialmente sfrutto decisamente meglio i periodi di I/O → inoltre in questo caso i tempi di attesa di P2, P3 sono decisamente più bassi!! → il tempo di attesa medio risulta  $(4+22+24)/3 = 16,66 \rightarrow 34 - 12 = 22$  e  $36 - 12 = 24$

→ anche in questo abbiamo un periodo in cui la CPU non risulta utilizzata → in quanto non ci sono processi pronti → finito I/O → torna poi in esecuzione!

→ dunque anche in questo caso vediamo che eseguire prima i processi corti rispetto a quelli più lunghi porta a migliori tempi di attesa medio !

### Scheduling Shortest-Job-First(SJF)

Si sfrutta l'idea di portare prima in esecuzione i processi che durano meno!

→ Si associa quindi a ogni processo la lunghezza del prossimo **CPU burst** → si usano quindi queste lunghezze per schedulare il processo con il minor tempo

+Ci sono due schemi possibili:

- nonpreemptive → quando la CPU è data a un processo e non gli può essere tolta fino alla terminazione del CPU burst ! → per cui anche se entra un processo più corto non rilasciamo la CPU!

- preemptive → se arriva un nuovo processo la cui lunghezza del CPU burst è minore del tempo rimanente al processo in esecuzione, questo nuovo processo entra in esecuzione! → se dunque arriva un processo che dura meno di quello in esecuzione allora questo viene portato in esecuzione!

→ Questo schema è conosciuto come Shortest-Remaining-Time-First (SRTF) → si

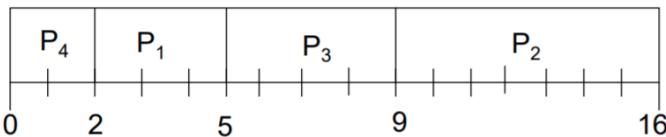
considera infatti principalmente si considera quanto tempo di CPU rimane da eseguire

→ SJF è ottimale → produce il minimo tempo medio di attesa per un insieme di processi fissati!

### +Esempio di SJF

| Processo | Tempo di arrivo | Tempo di Brust |
|----------|-----------------|----------------|
| $P_1$    | 0.0             | 3              |
| $P_2$    | 0.0             | 7              |
| $P_3$    | 0.0             | 4              |
| $P_4$    | 0.0             | 2              |

→ Con questa modalità si decide quindi di mettere in esecuzione inizialmente il processo  $P_4$  che dura solo 2ms! → in sequenza quindi si esegue  $P_1, P_3, P_2$



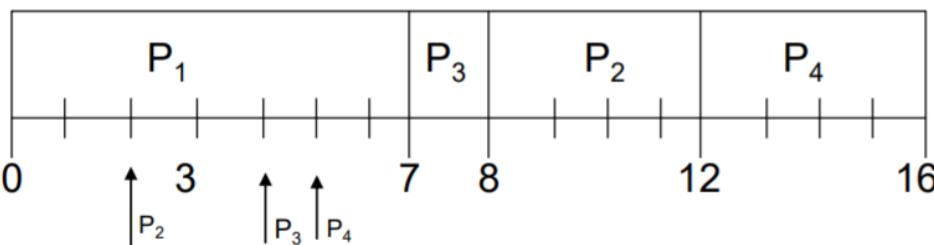
→ calcolando quindi il tempo di attesa medio ottengo  $(2+9+5+0) = 16/4 = 4\text{ms}$

→ e nel caso avessi usato una schedula FCFS otterrei invece (calcolando sempre il tempo di attesa medio) →  $(0+3+7+14) / 4 = 27/4 = 6,75\text{ms}$  → maggiore rispetto al caso precedente!!

+ Consideriamo quindi il caso non-preemptive (ma SJF)

| Processo       | Tempo di arrivo | Tempo di Brust |
|----------------|-----------------|----------------|
| P <sub>1</sub> | 0.0             | 7              |
| P <sub>2</sub> | 2.0             | 4              |
| P <sub>3</sub> | 4.0             | 1              |
| P <sub>4</sub> | 5.0             | 4              |

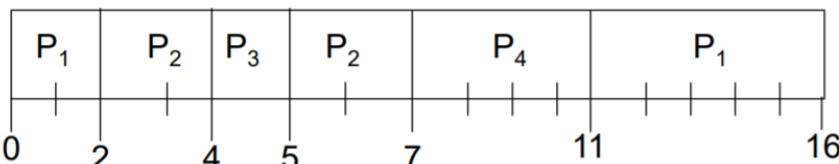
→ dove si considera anche il tempo di arrivo! → per cui al tempo zero essendoci solo il processo P<sub>1</sub> → questo entra subito in esecuzione e dura per tutto il suo tempo di Burst (ovvero 7ms) → anche se durante l'esecuzione di questo arrivano gli altri processi



→ finito quindi P<sub>1</sub>, si sceglie il processo a durata minore → P<sub>3</sub> → e successivamente P<sub>2</sub> e P<sub>4</sub> (a seconda di chi arriva prima)

→ calcolo quindi il tempo di attesa medio →  $(0+6+3+7) = 16/4 = 4\text{ms}$  → si calcola il tempo di attesa a partire da quando il processo arriva → per cui ad esempio per P<sub>2</sub> ho  $8-2=6\text{ms}$

+ Vediamo cosa succede nella soluzione preemptive → usando sempre gli stessi processi



→ eseguito inizialmente P<sub>1</sub> → arriva P<sub>2</sub> che minor tempo di durata (o tempo rimanente) → P<sub>1</sub> entra quindi in attesa e passa all'esecuzione di P<sub>2</sub> → allo stesso modo all'istante 4 → succede la stessa cosa con P<sub>3</sub> → infatti P<sub>2</sub> ha eseguito 2ms e gliene mancano 2 mentre P<sub>3</sub> gliene manca solo P<sub>3</sub>! → si esegue quindi P<sub>3</sub> → al tempo 5 arriva anche P<sub>4</sub> ma dura 4ms quindi si finisce il processo P<sub>2</sub> che era rimasto sospeso → finito questo passa all'esecuzione P<sub>4</sub> e infine P<sub>1</sub> finisce la sua esecuzione rimasta in sospeso

→ calcolo quindi il tempo di attesa medio (rispetto sempre la sequenza P<sub>1</sub>, P<sub>2</sub>... con i rispettivi tempi di attesa) →  $(9 + (2+1 - 2) + (4 - 4) + (7-5)) / 4 = (9+1+0+2)/4 = 12/4 = 3\text{ms}$

→ in particolare i tempi di attesa corrispondono quindi al tempo di prelazione → ovvero i tempi di durata dei processi che entrano mentre è in esecuzione un altro processo (ad esempio per P2 → gli entra in esecuzione il processo P3 che ha durata 1ms! (questo ovviamente considerando il tempo di arrivo!)

→ processi che arrivano successivamente possono *prelazionare* processi attualmente in esecuzione!

Predizione della lunghezza del prossimo CPU burst

+Come determinare la lunghezza del prossimo CPU Burst??

→ possiamo infatti sapere cosa sia successo solo in passato, ma sapere quanta CPU userà un processo da un certo momento al prossimo I/O → possiamo quindi stimare il prossimo CPU burst! → basandosi sulla storia passata dell'uso della CPU da parte dei processi si cerca di prevedere l'uso futuro della CPU!

→ in particolare si fa una media esponenziale dei tempi di CPU burst da parte del processo Presi quindi:

1.  $t_n$  = lunghezza dell' $n^{\text{esimo}}$  CPU burst
2.  $\tau_{n+1}$  = valore predetto per il prossimo CPU burst
3.  $\tau_0$  = valore iniziale preimpostato
4.  $\alpha, 0 \leq \alpha \leq 1$
5. Si definisce:  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$ .

→ al punto 5 si definisce quindi il valore predetto → ottenuto come media della lunghezza effettiva del n-esimo CPU burst e del valore predetto in precedenza → per cui più alpha sarà vicino ad 1 e più si tiene conto del valore attuale (altrimenti per alpha vicino a zero conta di più il passato!) → nei casi estremi:

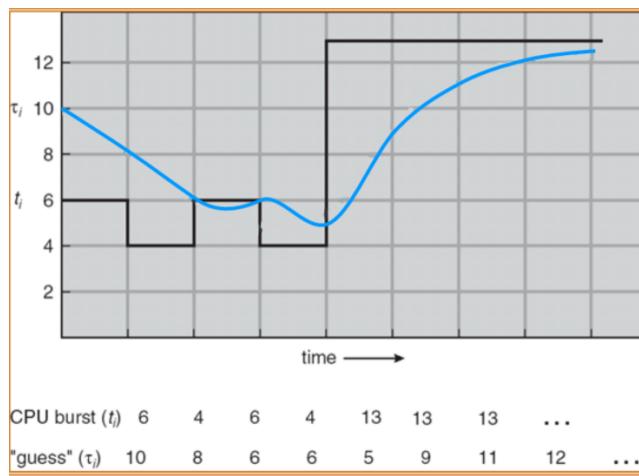
- alpha=0 •  $\tau_{n+1} = \tau_n$  → La storia recente non conta!
- alpha =1 •  $\tau_{n+1} = t_n$  → Conta solo l'ultimo CPU burst! → si ipotizza che il prossimo sia uguale a quello attuale!

+Espandendo la formula avrei:

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

→ ma siccome sia alpha che (1-alpha) sono tutti minori o uguali a 1 → ogni termine avrà peso inferiore rispetto al suo predecessore → via via i termini successivi contano sempre meno!

+Esempio:



- in nero è presente l'uso della CPU da parte del processo → passa infatti da 6 a 4 ripetutamente per poi usarne 13 ad ogni istante successivo
- in blu invece c'è la stima dunque il valore di  $\tau_i(n)$  → partendo dal valore 10 → il successivo risulta come media come  $10 + 6 = 16/2 = 8$ ! → allo stesso modo per l'istante successivo  $8 + 4 = 12/2 = 6$ ! → e così via → piano piano avvicinandosi sempre di più al valore effettivo!

### Scheduling con priorità

Si associa ad ogni processo un numero che indica la sua **priorità** → in modo che processi a priorità più alta siano eseguiti prima di quelli a priorità più bassa...

→ di solito si considerà l'opposto ovvero processi associati a interi più piccoli hanno priorità più alta! (abbastanza arbitrario)

→ anche in questo caso si può applicare lo schema preemptive o non-preemptive → ricordo che con preemptive si indica la possibilità che all'arrivo di un nuovo processo si tolga la CPU al processo in esecuzione e si porta in esecuzione quello appena arrivato! → mentre in quello non-preemptive invece qualsiasi processo arrivi successivamente aspetta la fine dell'esecuzione del processo attivo!

→ in particolare il metodo visto prima (SJF) può essere visto come uno scheduling a priorità dove la priorità è il tempo predetto del prossimo CPU burst

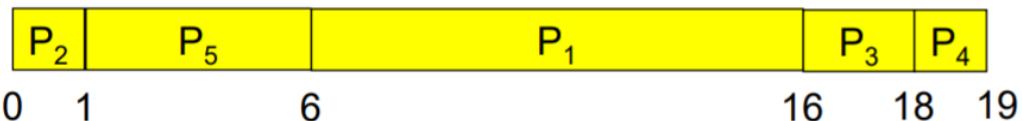
+In questo metodo però può capire un problema → *starvation* → processi a bassa priorità possono non essere mai eseguiti! → rimanere nella coda ready se continuamente arrivano processi a più alta priorità! → "muoiono di fame" in attesa di avere della CPU

→ una soluzione può essere l'uso della politica di *aging* → si usa una politica di modifica di priorità al passare del tempo → se dopo un certo tempo che il processo è sempre in coda → allora questo aumenta (di poco) la sua priorità → prima o poi quindi arriverà a priorità massima se rimane sempre in coda ready! → e quindi sarà eseguito!

+Esempio

| Processo | CPU | Priorità |
|----------|-----|----------|
| $P_1$    | 10  | 3        |
| $P_2$    | 1   | 1        |
| $P_3$    | 2   | 4        |
| $P_4$    | 1   | 5        |
| $P_5$    | 5   | 2        |

→ processi arrivati tutti a tempo zero → per cui la selezione si baserà solo sulla priorità! (che sia con o senza prelazione è lo stesso!)



→ mi calcolo quindi il tempo medio di attesa che risulta  $(6+0+16+18+1)/5 = 41/5 = 8,2$   
 → se invece avessi usato lo SJF otterrei  $(9+0+2+1+4)/5 = 16/5 = 3,2$  → che è minore!

+Nel caso di processi a stessa priorità allora si sceglierà in base all'ordine di arrivo (non sulla durata della CPU) → oppure altri scelte definite successivamente

### Scheduling Round Robin

Ogni processo prende una piccola unità di tempo di CPU (*time quantum*), solitamente 10-100 millisecondi. (quantità dunque non eccessivamente ridotta!)

→ Dopo che questo tempo è passato il processo viene interrotto e aggiunto alla fine dei processi pronti (anche se non ha finito la sua esecuzione!) → per cui anche questo è uno schema con prelazione

+ Se ci sono n processi nella coda dei processi pronti e il *quanto* di tempo è q → allora ogni processo prende  $1/n$  del tempo della CPU in parti di al massimo q unità di tempo alla volta.  
 → Per cui nessun processo aspetta più di  $(n-1)q$  unità di tempo! (prima di essere portato in esecuzione!) → questo quindi ci garantisce un certo tempo di risposta! → il processo entra in esecuzione dopo  $(n-1)q$  unità di tempo! → tempo che un processo nella coda ready dovrà aspettare per poter entrare in esecuzione

+nel caso di q molto grande → questo sistema diventa simile al FCFS → in quanto comunque diventano pronti rientrano in coda (una coda FIFO)

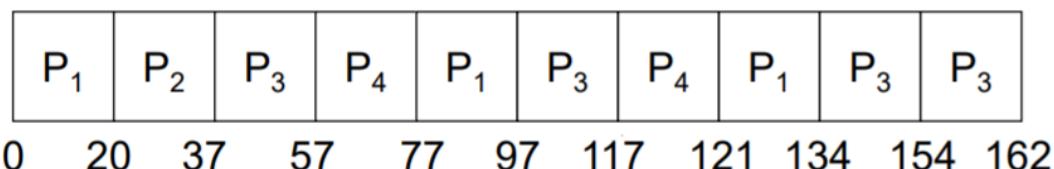
→ viceversa nel caso di q piccolo → abbiamo problemi → in quanto q deve essere grande rispetto al context switch, altrimenti ci sarebbe un tempo di *overhead* troppo elevato al cambio di contesto → se ad esempio il cambio di contesto dura 2ms → metà del tempo la spendo a fare il context switch... non va bene!

+Esempio:

### Processo      Tempo di Burst

|       |    |
|-------|----|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

(tempo di CPU in cui nessun processo fa I/O) → usando dunque un quanto di tempo di 20 ms ottengo un diagramma di questo tipo:



→  $P_1$  usa interamente il suo quanto → non finendo il tempo di burst entra in coda → per cui va in exec  $P_2$  → il quale finendo prima non consuma tutto il suo quanto di tempo → si passa quindi a  $P_3$  che come  $P_1$  non finisce il suo tempo e quindi va in coda → stessa cosa per  $P_4$  → finito il primo quanto di  $P_4$  si ritorna al primo processo in coda ovvero  $P_1$  → si passa dunque a  $P_3$  (dato che  $P_2$  ha già finito) e quindi  $P_4$  che stavolta finisce → finisce quindi anche  $P_1$  e rimane  $P_3$  con la sua parte rimanente

→ si passa quindi al calcolo del tempo medio di attesa

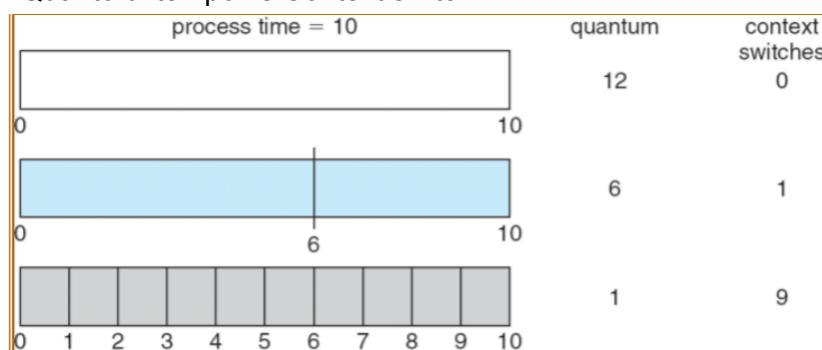
$$(((77-20)+(121-97))+20+(37+(97-57)+(134-117))+(57+(117-77)))/4 = (81 + 20 + 94 + 97) / 4 = 73 \text{ (mentre con SJF è 38)}$$

→ il tutto è ancora fattibile senza I/O → nel caso di I/O infatti diventa ancora più difficile → lo schema di esecuzione ci garantisce però che l'esecuzione prosegue per tutti i processi → ovvero non si esegue solo un processo e gli altri sono fermi ad aspettare! → a tutti viene garantito di essere in qualche modo eseguiti!

→ schema di scheduling usato in qualche modo per i processi interattivi! → per applicazioni che devono rispondere in tempi ragionevoli → si nota infatti che il tempo di risposta dei vari processi (ovvero la prima istruzione eseguita) è molto breve → 0 per  $P_1$ , 20 per  $P_2$ , 37 per  $P_3$  e 57 per  $P_4$  anche se i tempi di completamento sono in genere molto lunghi! → basta vedere quanto ci vuole per eseguire  $P_4$ !

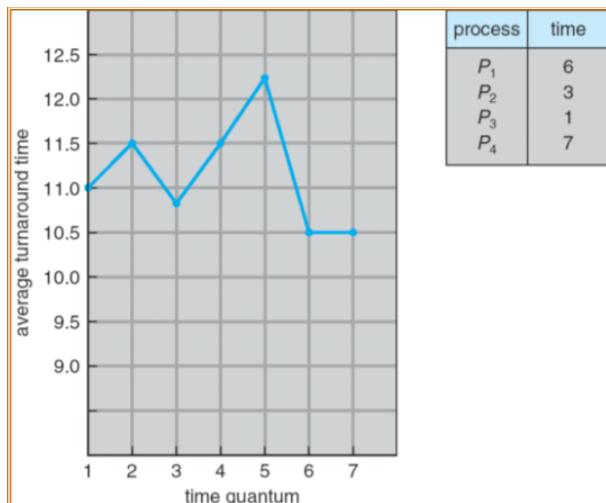
## 23/04/2021

+Quanto di tempo vs Context switch



→ maggiore sarà il quanto di tempo e minore è il numero di context switch! → infatti come abbiamo detto per quanto di tempo molto grande il Round Robin opera come un FCFS (zero context switch) → mentre per un quanto di tempo ad esempio di 1ms abbiamo ben 9 context switch!

+Esiste anche un collegamento fra la durata del quanto e il tempo di completamento medio per una serie di processi eseguiti con il Round Robin



→ si nota che all'aumento del time quantum il tempo medio di completamento non per forza migliora ma può essere anche peggiore! → sicuramente ad un determinato quanto di tempo abbastanza grande si passerà alla modalità FirstComeFirstServed → oltre un certo limite questo si stabilizza!

### Code multi-livello

La coda dei processi **pronti** è divisa in code separate → dunque non necessariamente abbiamo uno stesso algoritmo di scheduling in ogni coda ma si possono avere diversi algoritmi di scheduling!

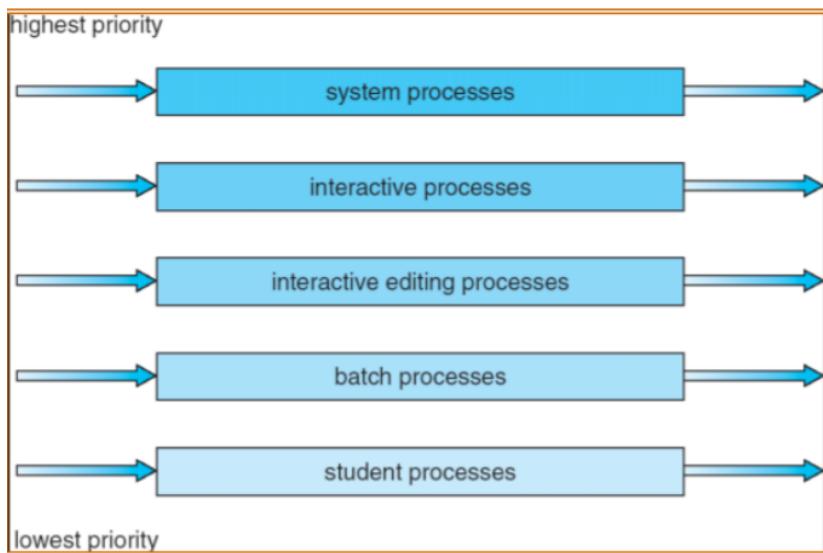
→ Ad Esempio si può avere la coda dei processi:

- *foreground* (interattiva) → i quali usano buoni algoritmi di scheduling per applicazioni interattive come il round-robin
- *background* (batch) → in quali non hanno necessità di una risposta immediata → per cui si possono usare altri criteri come il FCFS!

+Lo scheduling deve essere fatto tra le code → si decide quali processi mettere sulla CPU sulla base degli scheduling fra le code! → si possono anche qui avere diverse possibilità:

- Scheduling a priorità fissa → ad esempio la prima(foreground) è a più alta priorità e la seconda(background) a più bassa priorità → per cui finchè ho processi ad alta priorità si fa lo scheduling dei processi della prima coda (secondo l'algoritmo previsto per la coda) e solo quando la coda rimane vuota si passa a fare lo scheduling dei processi nella seconda coda (con un algoritmo diverso) → Possibile starvation!!
- Time slice (simile al round robin ma fra le code!) → ogni coda prende un certo tempo di CPU → e in quel tempo preciso viene eseguito lo scheduling di quella coda → nel momento in cui scade il quanto di tempo per quella coda si passa a schedulare i processi sull'altra coda! → ad esempio si può dare l'80% a foreground in RR e 20% per background in FCFS

+Esempio:



→ in questo caso ho un sistema con 5 code diversi → dove lo scheduling avverà inizialmente per i processi di sistema che hanno più alta priorità → e se solo nella coda non sono presenti dei processi → si passa alla coda successiva → come al solito può presentarsi il problema della starvation → soprattutto per la coda dei processi student!

+Per questo motivo sono presenti soluzioni alternative → come code multi-livello con retroazione! → ci sono dei criteri per i quali i processi possono passare da code a bassa priorità a code a più alta priorità in modo da garantire l'esecuzione di tutti i processi!

Si definiscono quindi un serie di parametri:

- Numero di code
- Algoritmo di scheduling per ogni coda
- Metodo usato per determinare quando promuovere un processo → come determinare quando portare una coda a bassa priorità ad una coda a + alta priorità!
- Metodo usato per retrocedere un processo
- Metodo usato per determinare in quale coda un processo deve entrare quando il processo necessita di un servizio

+Esempio:

abbiamo 3 code → Q0 – RR con quanto di tempo di 8 millisecondi

- Q1 – RR con quanto di tempo di 16 millisecondi • Q2 – FCFS

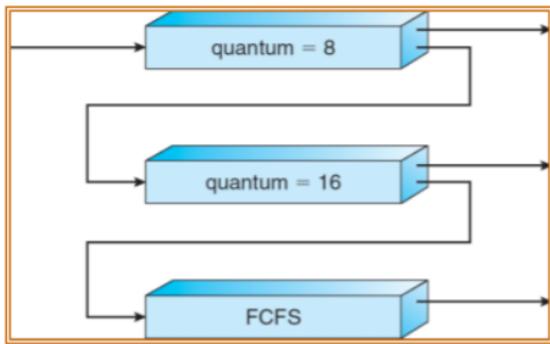
→ si decidono quindi dei criteri per il passaggio da una all'altra → Un nuovo lavoro entra sempre nella coda Q0 e quando ottiene la CPU → il lavoro riceve 8 millisecondi.

→ Se dunque il processo non finisce in 8 millisecondi, il lavoro e' spostato nella coda Q1 a più bassa priorità ! → In Q1 il lavoro riceve 16 millisecondi. Se non viene completato, viene tolto dall'esecuzione e mandato nella coda Q2!

→ In questo modo lo schema tende ad adattarsi al comportamento del processo! → ad esempio nel caso di processi interattivi (che usano poco la CPU) → allora rimangono sempre nella coda ad alta priorità! → per cui processi che usano poca CPU e fanno tanta interazione con l'utente tramite I/O

→ nel caso intermedio ci saranno i processi che usano un po' più di CPU (più di 8ms ma sempre meno di 16ms)

→ e infine ci stanno i processi che usano più di 16ms di CPU fra un I/O e l'altro



→ in questo caso non è previsto il percorso inverso → dalla coda FCFS non si può ritornare in alto ad essere molto interattivo! (si possono comunque aggiungere certi criteri per fare questo passaggio → ad esempio nel caso si abbia ad un certo punto dei processi che si comportano in modo più interattivo si passerà da processi FCFS a processi della prima coda! → con dei criteri che quindi portano i processi a fare questo passaggio → in modo che abbiano una maggiore priorità!)

### Percentuale CPU

Fissato un intervallo temporale viene calcolata la percentuale di tempo in cui le CPU sono state in uso → ovvero sono state utilizzate → ad esempio per 1 secondo si stabilisce che la CPU è stata usata al 20% → il 20% la CPU è stata utilizzata per il restante 80% la CPU era in attesa di essere in esecuzione → non c'era nessun processo che doveva essere eseguito!

→ è questo Un parametro molto usato per indicare l'uso della CPU da parte dei processi → in particolare è presente un percentuale che indica l'uso di CPU globale → ma si può avere un parametro specifico di un preciso per avere informazioni su quanto quel processo è stato usato dalla CPU durante quell'intervallo temporale → ad esempio se un processo usa il 100% di CPU → significa che la usa tutta lui → oppure nel caso multicore una CPU è usata interamente da un singolo processo! (o addirittura più del 100% → processo usa più core) → l'uso della CPU si può dividere facilmente tra i processi che sono in esecuzione e in particolare una CPU al 100% significa che è sempre in uso!  
+Infine nel caso di sistemi server si tende a tenere sotto l'80% il carico sulla CPU → nel caso si superi questo limite si divide il carico su più server attivi contemporaneamente per gestire il lavoro di queste richieste che vengono date!

### Carico medio CPU

→ nei sistemi unix è presente un altro parametro → detto carico medio di CPU e indica la capacità di eseguire i processi in modo regolare senza eccessivi ritardi  
→ Viene valutato il carico istantaneo (in un certo momento) che include il numero di processi in esecuzione e in attesa di essere in esecuzione! → dunque ci dice la lunghezza della coda ready → dei processi pronti o in esecuzione  
→ su linux include oltre il numero di processi in stato **TASK\_RUNNING** (in esecuzione o ready) anche quelli in stato **TASK\_UNINTERRUPTIBLE** (in attesa di I/O sul disco) → parametro di carico dunque a livello di sistema  
→ il parametro che dunque viene mostrato all'utente sarà la media esponenziale di questo carico → campionato ogni 5 s → media che viene calcolato a 1min, 5min e 15 min

→ preso quindi  $na(t)$  = "numero di task attivi al tempo  $t$ " → task che sono attivi e in attesa su linux → dunque il carico medio sarà:

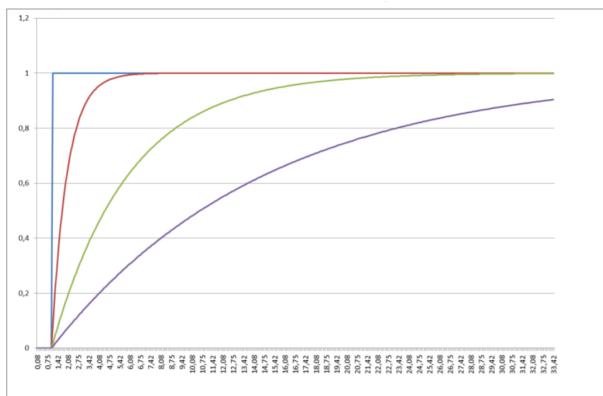
■ **avgLoad( $t$ )** =  $\alpha \text{ avgLoad}(t-1) + (1 - \alpha) na(t)$

→ dunque con peso alpha si basa sul carico medio precedente e  $(1-\alpha)$  il numero di task attivi al momento → il valore di alpha invece varia a seconda del momento in cui calcolato:

- $\alpha = 1/e^{(5/60)} = 0.92$  per media a 1 minuto
- $\alpha = 1/e^{(5/(60*5))} = 0.9834$  per media a 5 minuti
- $\alpha = 1/e^{(5/(60*15))} = 0.994$  per media a 15 minuti

→ nel caso di sistema multicore avremo un risultato positivo se questo valore risulta inferiore ad 1 → infatti vorrà dire che in media ho meno di un processo in attesa! → il sistema non è fermo...

+Viene ad esempio fuori un grafico del genere: (zoomma per vedere meglio)



→ dove in blu ci sarà il numero di task attivi al tempo  $t$  → che appunto sarà zero fino a  $t = 0,75$  e 1 successivamente! → ovvero ci sarà almeno un task attivo → successivamente c'è il carico medio (rispettivamente) per 1,5 e 15 minuti → per cui considerando quello che avviene nell'ultimo minuto questo si avvicina di più al numero di task attivi rispetto a considerare ciò che è avvenuto negli ultimi 5 o 15 minuti! → soprattutto nell'ultimo caso influiscono tutti gli zero che ci sono stati nei precedenti 15 minuti → cresce più lentamente!

+Esempio:

```
ubuntu@DESKTOP-HSMGJO0: /mnt/c/Users/pierf/process
1 [] 6.2% 4 [] 1.3% 7 [] 0.0% 10 [] 100.0%
2 [] 0.0% 5 [] 0.0% 8 [] 0.0% 11 [] 0.7%
3 [] 100.0% 6 [] 0.0% 9 [] 0.0% 12 [] 0.0%
Mem[|||||] 2.20G/24.9G Avg[|||||] 17.4%
Swp[0K/7.00G Tasks: 12, 31 thr, 1 kthr; 3 running
Load average: 1.91 0.96 0.38
Uptime: 16:21:56

PID: MAJFLT: DISK I/O/W: OOM:USER: PRI: NI: VIRT: RES: >SHR: CPU%: NICE: TIME+: NIWP: Command:
1 0 no perm 0 root 20 0 896 576 516 S 0.0 0.0 0:00:03 2 /init
287 0 no perm 0 root 20 0 896 80 20 S 0.0 0.0 0:00:04 1 /init
288 0 no perm 0 root 20 0 896 80 20 S 0.0 0.0 0:00:04 1 /init
289 0 0.00 B/s 0 ubuntu 20 0 10168 5324 3484 S 0.0 0.0 0:00:15 1 /init
375 0 0.00 B/s 0 ubuntu 20 0 8280 3824 3048 R 0.0 0.0 0:00:14 1 bash
374 0 0.00 B/s 0 ubuntu 20 0 2492 516 452 R 99.8 0.0 2:31.92 1 htop
372 1 0.00 B/s 0 ubuntu 20 0 2492 584 526 R 99.8 0.0 3:39.41 1 ./cupbound
97 0 no perm 0 root 20 0 896 80 20 S 0.0 0.0 0:00:00 1 ./cupbound
106 0 no perm 0 root 20 0 896 80 20 S 0.0 0.0 0:00:01 1 /init
107 12 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:02.79 14 /init
147 0 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:15 14 /init
146 0 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:28 14 /init
142 0 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:08 14 /init
125 0 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:23 14 /init
124 1 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:23 14 /init
123 1 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:26 14 /init
122 6 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:11 14 /init
121 1 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:26 14 /init
112 0 no perm 0 root 20 0 1646M 32236 14424 S 0.0 0.1 0:00:00 14 /init
F1/help F2/Setup F3/Search F4/filter F5/sorted F6/column F7/lice F8/lice F9/fill F10/exit
```

→ si sono lanciati dal prompt due volte lo stesso programma il quale sfrutta al 100% la CPU e si nota inizialmente che sono stati impiegati 2 core al 100% e inoltre (tutto questo grazie al comando htop) guardano il load average → si avvicina piano piano al valore di 2 (ovvero esattamente 2 processi attivi) → quello calcolato a 1min ovviamente ci mette di meno e quello a 15 minuti più di tutti!

→ per cui se questi tre numeri sono in crescita significa che c'è stato un aumento di carico  
→ per cui se questi rimangono poi fissi a valori molto alti significa che il carico sul computer è molto alto e dunque può portare a dei problemi! → in particolare nei sistemi unix poniamo il vincolo per cui il load average deve essere inferiore al numero di core → ad esempio se in questo caso supera il valore di 13-14 significa che non abbiamo abbastanza CPU e il sistema va in sovraccarico!

(notare che nel caso di sistemi linux il carico medio tiene conto anche dei processi che sono in attesa di qualche input)

+In realtà sto eseguendo tutto su una macchina virtuale...



→ per cui vedendo l'utilizzo complessivo della macchina in realtà sembra andare tutto in modo uniforme → notare nel grafico viene indicato il tempo di esecuzione in modalità utente e in modalità kernel!

+Andando a uccidere i processi con il comando kill (sempre dalla macchina virtuale) → si vedrà abbassare i valori del load average

+Esercizi sullo scheduling(da rivedere da min38 fino alla fine)

# 28/04/2021

## Scheduling Linux

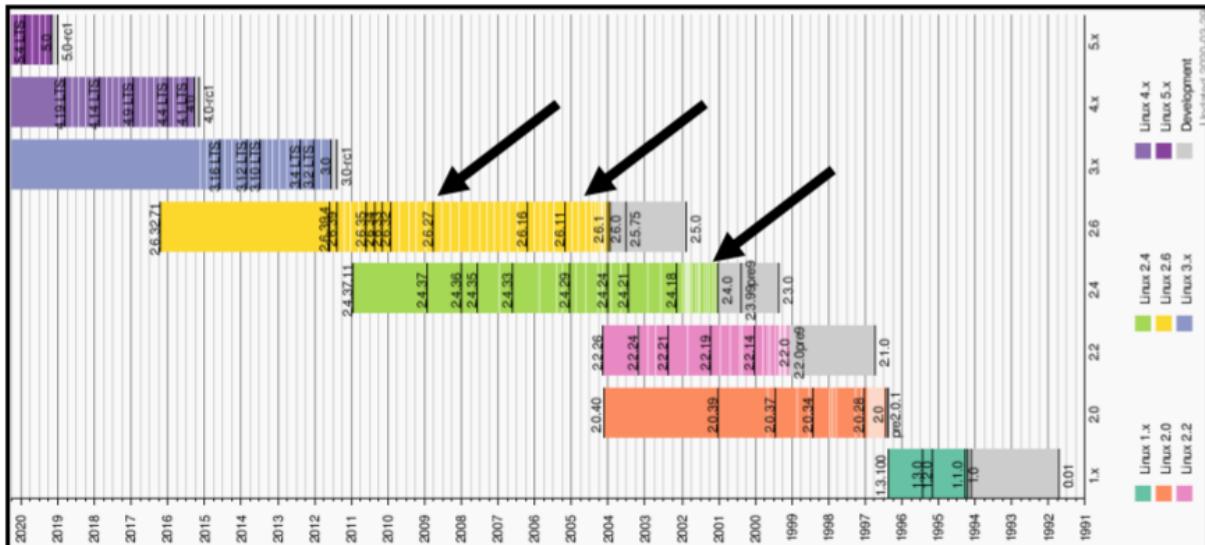
Come linux gestisce l'esecuzione dei processi... il kernel di linux è passato in varie versioni da una basilare e semplice e versioni più complesse dato che il sistema linux risulta molto adattato in diversi ambienti molto diversi fra loro → dai telefoni con android i quali hanno un kernel linux ai pc desktop → sistemi con necessità diverse di esecuzione delle applicazioni! → adattarsi a tutti gli ambienti di esecuzione ha portato all'evolversi del modo nel quale linux gestisce l'esecuzione delle applicazioni!

Linux è un kernel OpenSource e partì con gli obiettivi di:

- Gestire le applicazioni in *timeSharing*, condividendo il tempo di esecuzione delle CPU fra i vari processi
- Avere una gestione dinamica delle priorità! → il sistema poi la priorità in base da parte dei processi e anche poi la possibilità di cambiare poi la priorità dei processi
- Avere tempi di risposta brevi → dunque le applicazioni rispondono velocemente → i sistemi sono usati dagli utenti non come sistemi batch e quindi hanno bisogno che rispondano con tempi abbastanza brevi
- devono anche gestire la cpu in background → per processi intensi quindi in questo caso avere un throughput elevato → dunque avere possibilità di eseguire tanti processi e produrre dei risultati in modo veloce...
- evitare la starvation → dunque evitare che i processi non acquisiscono più la CPU → ovvero rimangano in attesa della CPU infinitivamente
- Gestire processi (soft) real-time → cioè la cui correttezza dipende dal tempo in cui vengono eseguiti → in quanto devono effettuare certe operazione i determinati vincoli temporali → ad esempio un player multimediale deve mandare i campioni da riprodurre nella scheda audio entro determinati tempi, altrimenti all'esecuzione del player si sentiranno dei glitch → piccoli problemi per non aver rispettato i vincoli temporali (non catastrofico) → se non riescono ad essere seguiti entro un certo tempo questi processi falliscono!

+Prevede inoltre uno scheduling con prelazione su codice utente → per cui se siamo in esecuzione in modalità utente (dunque nel codice dell'utente) si ha la prelazione → ma dal kernel 2.6 anche il codice del kernel supporta la prelazione, per cui durante l'esecuzione durante una chiamata di sistema, quel determinato processo potrà essere prelazione per dunque passare l'uso della CPU ad un altro processo! (rimanendo a mezzo di una chiamata di sistema) → prima di questo non era possibile! → per poter interrompere una chiamata di sistema bisogna prima arrivare ad un punto in cui fosse sicuro rilasciare la CPU! (evitando problemi alla gestione di strutture dati del kernel!)

+timeline delle varie versioni di linux



+ Alcune info inutili:

- Kernel 2.4 (2001) – Scheduler basato su *epoch* → nel quale si guardavano TUTTI i processi pronti per stabilire poi quale dovesse essere messo in esecuzione → scheduler O(n) con n numero dei processi
- Kernel 2.6.8.1 (2004) – Scheduler O(1) → tempo per scegliere quale processo mettere in esecuzione risulta essere costante! (e non dipende dal numero di processi che ci sono nel sistema)
- Kernel 2.6.23 (2010) – Completely Fair Scheduler → modo di fare scheduling utilizzato attualmente!
- Kernel 3.9 (2013) – Scheduler Earlier Deadline First per task periodici → scheduling più apposito per processi periodici → si basa infatti sapendo quando i processi periodici dovranno essere eseguiti, mettendo dei vincoli sul tempo di esecuzione  
→ per cui risulta esserci uno scheduler preciso per gestire questo tipo di processi

+ In realtà linux implementa la specifica *posix* per lo scheduler → la quale è più generale anche per altri kernel unix... per cui implementa linux implementa una specifica posix

→ si basa su una priorità *statica* e una *dinamica*:

priorità statica ha un range che va da 1 – 99 (con 99 priorità max) usata per scheduling dei task (detti soft) real-time

+ con priorità statica i task possono essere schedulati secondo priorità:

- SCHED\_FIFO: quanto di tempo illimitato → gli viene lasciato tutto il tempo che è necessario → lasciano la CPU solo se si bloccano per I/O o altro evento, oppure se terminano, oppure ancora se un task a più alta priorità diventa pronto! → se avessimo un processo a più alta priorità (ad esempio 99) di tipo FIFO → non può esistere un altro task di più altra priorità per cui se il task entra in loop non rilascia mai la CPU fino a che il sistema si bloccherà!
- SCHED\_RR: soggetti a scheduling Round-Robin (nella stessa coda dei task FIFO)  
→ viene dato un quanto di tempo predefinito al processo che è definito con questo tipo di scheduling  
→ processi round robin che sono comunque messi nella stessa coda dei task fifo, per cui l'unica differenza con lo schema precedente è il fatto di usare quanti limitati nel tempo

→ nei casi in cui si esegue il debugging di una applicazione che usa questo tipo di scheduling, allora una cosa che conviene fare è far partire una shell a priorità elevata in modo che sia possibile su questa shell, poter ripristinare lo schema nel caso questa si sia bloccata in una certa maniera... in modo da non dover spengere la macchina e farla ripartire! (questo perchè come prima se un task FIFO entra in loop infinito sulla (unica) CPU il sistema può bloccarsi!)

Poi è presente la priorità statica 0 (quella più bassa), dove sono presenti dei task i quali sono schedulati nella modalità *SCHED\_OTHER/SCHED\_NORMAL* → che viene usata per i processi normali... generalmente i processi partono tutti da priorità zero

→ in realtà hanno tutti la stessa priorità e però all'interno di questa priorità zero hanno una priorità dinamica → che dunque può cambiare!!

→ a questo livello qui in particolare cambia la politica di scheduling in base al kernel!! che verrà utilizzato → a versioni di kernel diverse avremo versioni di scheduling differenti!

+Inoltre abbiamo anche un *SCHED\_BATCH* → per processi a priorità 0 ma che non sono interattivi → in questo modo si dice esplicitamente al sistema : "occhio hai a che fare con un processo batch" dunque trattalo come tale (processo cpu intensive → che usano molta cpu)  
→ si dà in questo modo più priorità ad altri processi che magari risultano essere più interattivi! → ovvero non usano intensamente la cpu!

+Infine sono presenti quelli con *SCHED\_IDLE*, i quali sono quelli che sono eseguiti SOLO SE LA CPU RISULTA LIBERA! → se nessun altro processo usa la cpu allora può andare in esecuzione e appena qualcun altro vuole usare la cpu questa fermerà la sua esecuzione!  
→ risulta quindi fra quelle a priorità zero ma fra queste a priorità più bassa! → se quindi si deve decidere lo scheduler li mette in exec solo quando non c'è nessun altro che deve usare la cpu tra quelli dichiarati ad esempio other!

## Scheduling Linux – Kernel 2.4

Task tutti basati a priorità statica 0 e basato su 'epochi'... cosa sono??

→ Ad ogni processo durante l'epoca viene assegnato un quanto di tempo (che consuma durante l'epoca) → dove però ogni processo può avere una durata del quanto diversa! → per cui la priorità dinamica è legata alla durata del quanto → più è lungo il quanto di tempo (ovvero più assegno cpu al processo) e più avrà priorità (??)

→ Quando tutti i task in coda ready hanno terminato il loro quanto l'epoca è finita e vengono calcolati i quanti di tempo per l'epoca successiva, inoltre il quanto di tempo viene aumentato della metà di quanto rimasto dall'epoca precedente! → questo perchè in realtà un processo può non aver usato tutto il quanto → perchè andato in attesa di un I/O → dunque non ha usato tutto il quanto ma ormai è uscito dalla coda! → Per cui è come se venisse premiato di questo fatto dandogli un po' più di CPU! → avvantaggiando i processi interattivi o I/O bound! (rivedere processi I/O bound)

In realtà è possibile modificare la priorità in modo esplicito → l'utente può assegnare a piacere delle priorità sui processi aumentandola o riducendola avvantaggiando altri processi!  
→ si fa questo impostando il *nice value* [-20 ,19], dove i valori negativi aumentano la priorità del processo! (viceversa i positivi la diminuiscono → diminuendo il suo *timeslice*)

→ in questo caso solo il *superuser* può aumentare la priorità! (gli altri possono solo diminuirla → per lasciare la CPU ad altri processi)  
→ Per decidere quale task eseguire viene fatta scansione di tutti i task nella coda ready, per cercare il task con la *goodness()* migliore → ovvero una funzione che stabiliva in base a vari parametri (come priorità-nice value-uso di CPU ecc...), quale fosse il processo da mettere in esecuzione → per farlo faceva una scansione lineare della coda per cui porta ad avere un complessità di → O(n) → se avevo ad esempio 1000 processi me li dovevo guardare tutti e 1000 per decidere quale fosse il migliore

+L'altro vincolo era il fatto di avere un'unica coda per tutte le CPU! → per cui tutte le CPU per decidere quale processo mettere in esecuzione dovevano aspettare di poter accedere alla coda! → perchè accedere in modo concorrente ad una struttura dati, simile alla coda crea dei problemi! → per cui in realtà ci può accedere una CPU alla volta! → non molto efficiente per sistemi con più CPU!  
→ Infatti nel caso in cui CPU cercano di accedere insieme interferiscono l'una con le altre, rallentando l'esecuzione dei processi

→ Con l'aumento notevole del numero processi, e nel caso di multiprocessing simmetrico → ad esempio 20 CPU, queste stanno ad aspettare perchè la struttura dati era comunque unica e quindi il tempo di CPU viene perso... Per cui non si usa più questo tipo di scheduling

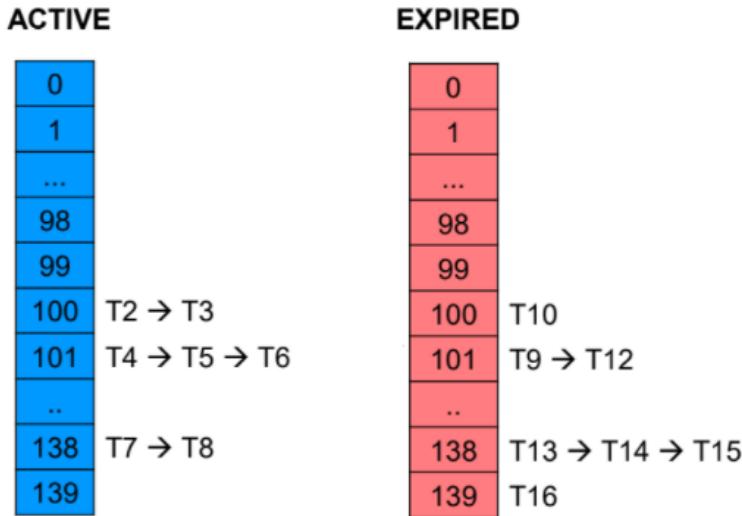
## Linux scheduler O(1)

Introdotto nel 2004 → in questo caso il tempo necessario a stabilire quale task mettere in esecuzione non dipende dal numero di task che sono in esecuzione!!  
→ Per cui viene fatto in tempo costante! (Da O(n) a O(1) )

+Sono presenti 140 livelli di priorità (con 0-139 e 0 priorità massima) dove:  
• 0-99 usati per *task real-time* → seguendo la politica FIFO/RR ecc...  
• 100-139 per task utente (120 + nicevalue del task che va appunto da -20 a +19) → per cui i vecchi processi a priorità zero vengono mappati in questo range! → qua dentro ci saranno dunque gli *sched\_other* o gli *sched\_normal* (i processi fifo o RR sono sopra)

+Per ogni livello di priorità **due** liste FIFO (non più una sola):  
• una con i task con quanto di tempo **non esaurito** (task attivi)  
• una con i task con quanto di tempo esaurito (*expired*)

+Ad esempio:



→ Al livello di priorità 100 abbiamo T2 e T3 che sono in attesa di CPU, successivamente ad una priorità più bassa → 101 ci sono T4,T5,T6 in attesa ecc...

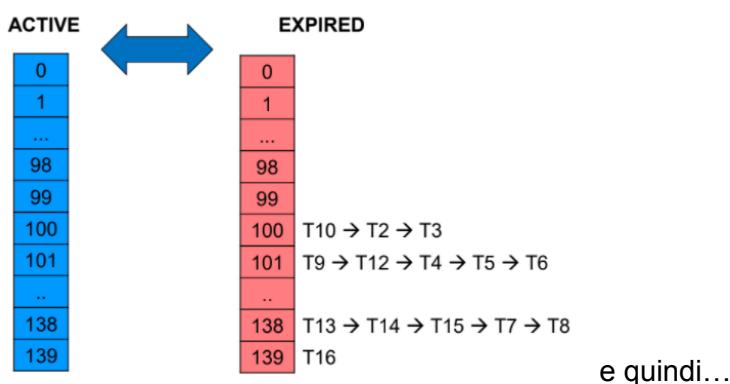
→ dall'altra parte invece nella coda expired, ci sono i task che hanno già usato il loro quanto di tempo e quindi sono in attesa del successivo round!

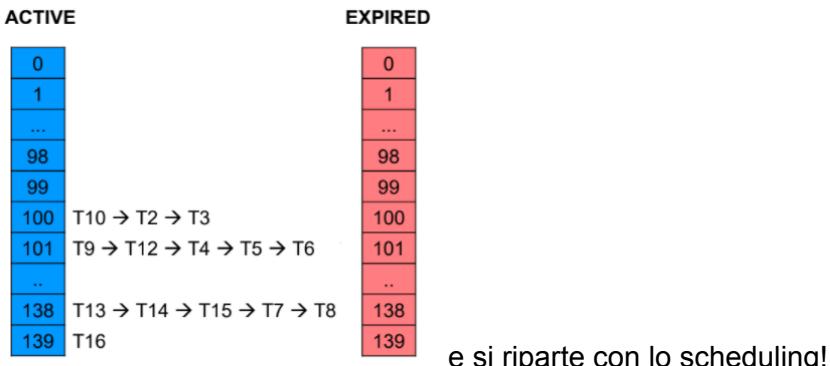
+ Come funziona lo scheduler quindi?

→ Sceglie il task in **testa** alla coda a priorità più **alta** (che poichè si prende dalla testa viene fatto in tempo costante!) → quando ha terminato il quanto viene messa nella corrispondente coda expired

→ successivamente quando tutte le liste dei task attivi sono vuote si scambiano con quelle expired! → la coda blu diventa expired, e l'altra active!

+ Trovare la priorità per cui esiste almeno un task attivo si fa in tempo costante! → infatti non dipende dal numero di task che sono presenti! → potrebbero esserci mille in 1 ma comunque ci entro in tempo costante! (implementato con un bit array) → O(1)





+Così descritto sembra essere uno scheduling con priorità fissa, dove il nice value rimane costante! → ad esempio il task 16, rimarrà sempre con priorità più bassa con possibilità di andare in starvation se non può entrare in esecuzione!

+Per avvantaggiare i task che usano I/O, questi vengono premiati e penalizzati quelli CPU bound con un incremento/decremento di priorità nel range [-5,+5]

→ questo non è valido per i task real-time!

→ l'incremento si basa sul calcolo dello *sleep\_avg* del task → il quale mi dice (in media) quanto tempo è stato in sleep → ovvero in attesa di I/O → indice che viene quindi incrementato fino ad un massimo → e infine verrà decrementato questo valore del tempo in cui ha usato la CPU!

→ il *bonus* si calcola quindi come:

$$\text{bonus} = (\text{sleep\_avg} * \text{MAX\_BONUS}) / \text{MAX\_SLEEP\_AVG} - \text{MAX\_BONUS}/2$$

( $\text{MAX\_BONUS} = 10$ )

→ più piccolo sarà lo *sleep\_avg* è più piccolo è il bonus! → per *sleep\_avg* = 0, il bonus viene -5! → bonus che porta ad un decremento di priorità!

+il bonus non può far passare task non-RT a priorità dei task RT (0-99)

→ e infine il timeslice dipende dalla priorità statica del task → maggiore priorità (valore più basso) timeslice più grande!

+Successivamente sono stati aggiunti anche degli *Interactivity credits* → dei bonus ulteriori... ottenuti quando task aspetta per tanto tempo!

→ che però vengono persi quando task usa tanta CPU!

→ per cui sono usati per non far perdere il bonus che un processo ha ottenuto solo perché un task I/O bound usa occasionalmente tanta CPU!

+Si risolve anche il problema precedente delle CPU multicore → ogni CPU ha la sua coda dei processi, i processi vengono spostati per bilanciare il carico delle CPU! → è possibile infatti spostare un processo da una cpu all'altra se è presente un processore troppo sovraccarico → allora si fa in modo di spostarlo ad un'altra CPU

+Anche questo tipo di scheduler è stato abbandonato per la sua complessità del codice gestione delle euristiche per bonus... che faceva perdere del tempo all'esecuzione, in quanto andavano aggiornati tutti i crediti insieme

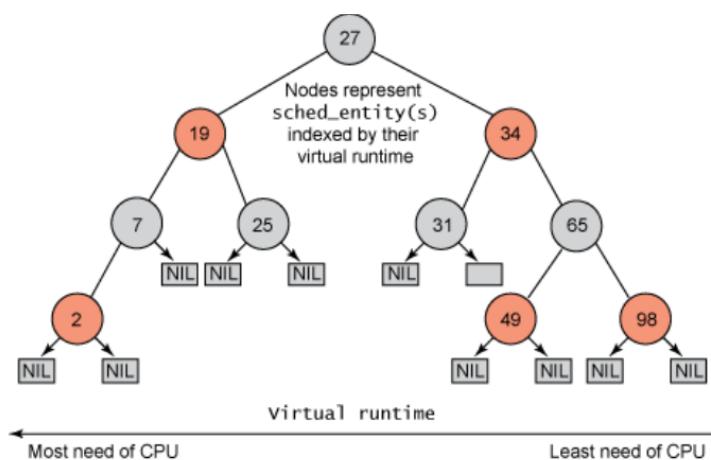
→ Per cui nel Kernel 2.6.23 (2010) è stato introdotto *Completely Fair Scheduler CFS*

## Linux Completely FairScheduler

Si basa sulla considerazione di essere quello giusto (?) rispetto a tutti... si basa sul fatto di assegnare la CPU a chi ha usato **meno** CPU → se deve decidere sulla base di quanta CPU hanno utilizzato e se c'è nè uno che ha usato meno CPU quello è avvantaggiato rispetto ad altri!

→ per decidere quale task eseguire nella coda ready viene selezionato quello con il ***virtual\_runtime minore*** ovvero quello che ha usato meno CPU!

+Per trovare tra un insieme di task che sono in attesa quello che ha usato meno CPU, si usa la struttura dati degli alberi binari rosso e neri! → i quali sono degli ab di ricerca bilanciati → i quali ci garantiscono che l'aggiornamento della struttura dati si faccia in un tempo logaritmico! sul numero di processi che ci stanno all'interno



→ ogni nodo rappresenta un processo in attesa e i processi più in basso a sinistra sono quelli che hanno usato meno CPU! → per cui nello scegliere fra tutti quelli da mettere in esecuzione viene scelto quest'ultimo! → per arrivarci si scorre tutti i dati sinistri trovando il minore di tutti e una volta trovato, questo cambierà il suo tempo di esecuzione e quindi si sposterà nell'albero in modo da essere messo nella giusta posizione (anche questo fatto in tempo logaritmico! →  $O(\log n)$ )

→ detta in questo modo sembra però che tutti abbiamo la stessa priorità... da qui il riutilizzo del *nice level* → il quale viene utilizzato per aggiornare il virtual real-time...

→ invece di aggiornarlo per il tempo effettivo usato per la CPU, questo tempo viene in realtà moltiplicato per un fattore che dipende dal nice value! → fattore  $(1.25 ^ \text{nice})$  → e ricordando che il nice value varia fra un valore di (-20 , 19) → per cui elevendo alla **meno** 20 nel caso di massima priorità, oppure alla +19 per indicare la minore priorità!

→ è come se la CPU fosse più veloce per i task ad alta priorità → in sostanza è come se gli venisse tolta una parte del tempo che lui non ha usato e quindi sarà preferito nell'uso della CPU successiva! → in quanto viene incrementato di poco il suo valore nell'albero! →

continuando a stare nella parte a sinistra dell'albero dove ci sono i valori minori → viceversa per i processi a più alta priorità accade l'opposto!! → rimangono nella parte destra addirittura scendendo nell'albero!

→ task i/o bound selezionati prima rispetto a quelli CPU bound → minore virtual-runtime

+Un task viene selezionato ed eseguito, quando verrà fermato verrà aggiornato il suo virtual\_runtime e se entra nella coda **ready** viene inserito nell'albero RB

→ I task I/O bound tendono ad usare poca CPU e quindi tendono ad essere selezionati prima di quelli CPU bound → i primi infatti usano poca CPU per cui l'aggiornamento li riporterà nella parte iniziale dell'albero, mentre quelli CPU bound scenderanno verso l'estremo destro

+Anche il quanto di tempo assegnato dipende dal nice level del task → il tempo assegnato viene dato in base ad un paramentro detto **peso** che si calcola come:

- weight =  $1024 / (1,25 ^ \text{nice})$

e per particolari valori si ottiene:

- nice = 0 → weight = 1.024
- nice = -1 → weight = 1.280 ... nice = -20 → weight = 88.817
- nice = 1 → weight = 820 ... nice = 19 → weight = 14

→ notare che per valori negativi, i pesi aumentano!

→ viceversa nei nice positivi i pesi diminuiscono fino al minimo con nice = 19

→ a quel punto il quanto viene calcolato sulla base di tutti gli weight degli **n** task nella coda ready!

- **scheduling\_period** =  $\max(n * \text{sched\_min\_granularity\_ns}, \text{sched\_latency\_ns})$

→ con sched\_min\_ecc... che è il minimo tempo di CPU assegnato ad un processo, dunque ad n grande risulta un grande periodo di scheduling, viceversa per n piccoli si sceglie lo sched\_latency\_ns in modo da evitare di avere tempi di scheduling molto bassi!

+Infine il time\_slice fra tutti i task che sono pronti viene diviso in base al peso che è stato calcolato! per cui per ogni task avrà:

- **time\_slice**( $T_k$ ) = **scheduling\_period** \* weight( $T_k$ )/(SUM weight( $T_i$ ))

→ Dunque all'aumentare della priorità (nice level basso) avrà un timeslice più grande rispetto agli altri task! (time\_slice → quanto di tempo)

In realtà la priorità si adatta, infatti i task I/O bound che usano poca CPU, tenderanno ad essere spesso nelle parti iniziali, ovvero avere valori bassi nella virtual\_runtime e quindi ad essere selezionati prima rispetto agli altri!

→ Ricordare che questo tipo di scheduling viene fatto in questo modo solo per i processi a priorità zero!

(scheduling per sistemi android usati oggi)

+Con il kernel 2.6.23 è stato riorganizzato il codice per lo scheduling ed è stata definita un interfaccia per algoritmi di scheduling

→ Questo permette di definire nel kernel più *classi di scheduling* ognuna gestita con il proprio algoritmo

→ Le classi sono poste in sequenza di priorità (in ordine decrescente):

- STOP, per fermare il lavoro su CPU
- RT, per processi real-time
- FAIR, per processi normali
- IDLE, per attività a bassissima priorità

→ In questo modo risulta più semplice aggiungere nuovi algoritmi di scheduling al sistema

+Link per chiurosità <https://github.com/torvalds/linux/tree/master/kernel/sched>

+Compito del 25-11-2020

## Sincronizzazione dei processi

Vediamo come gestire l'uso di dati condivisi da parte thread/processi in modo concorrente...

→ dati condivisi può portare ad avere dei dati inconsistenti, dovuto a scheduling con prelazione in processore, oppure comunque all'accesso in parallelo alla memoria in un sistema multiprocessore

→ Nel caso in cui avvenga appunto la prelazione da parte di un altro processo che deve accedere alla struttura dati condivisa... il fatto che venga interrotta un'azione a metà può portare poi a problemi...

→ compito del sistema operativo è quello di fare in modo di mantenere la consistenza dei dati tramite dei meccanismi per assicurare l'esecuzione ordinata dei processi/thread che risultano essere in qualche modo concorrenti!

+Supponiamo di realizzare un processo/thread *produttore* dei dati e un processo/thread consumatore, i quali condividono una zona di memoria per memorizzare degli elementi che fanno parte della coda??

→ il processo che genera, prende e inserisce nella coda mentre il consumatore prende e estrae dati dalla coda

→ si usa una variabile condivisa intera count che tiene traccia di quanti elementi siano pronti per il consumatore (è impostato inizialmente a zero) → il valore count verrà incrementato dal thread che *produce* → si mette il dato nella coda e poi incrementa mettendo un dato in più quando un nuovo elemento risulta disponibile viceversa sarà decrementato dal consumatore quando effettivamente consuma un elemento!

+Esempio di thread produttore:

```
while (true)
{
 /* produce un elemento e lo mette in
 nextProduced */
 while (count == BUFFER_SIZE)
 ; // do nothing
 buffer [in] = nextProduced;
 in = (in + 1) % BUFFER_SIZE;
 count++; }
```

→ dopo aver prodotto un elemento e averlo messo nella struttura dati, aspetterà se il count è pari alla lunghezza massima ovvero il buffer è pieno → bisognerà aspettare!

→ (si farà un wait ??)

→ quando dunque avrà un posto libero (magari liberato dal processo consumatore), identificato dalla variabile *in* → in questo posto verrà messo il valore prodotto e quindi aggiornata la posizione dell'*in* per andare verso la posizione successiva! (questo tramite un buffer circolare in modo che una volta superato il limite del buffer si ritorni alla posizione iniziale!) → infine si incrementerà il valore di count!

+Esempio thread consumatore

```
while (true)
{
 while (count == 0)
 ; // do nothing
 nextConsumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 count--;
 /* consuma l'elemento in nextConsumed */}
```

Allo stesso modo ma opposto, aspetta che sia presente qualcosa nel buffer → se il count è zero significa banalmente che il produttore non ha fatto nulla e quindi aspetta...

→ la cpu guarda il valore di questa variabile → nel momento in cui count != 0, allora il produttore sarà riuscito a inserire qualcosa → incrementa il count per cui si esce dal while  
→ si prende il valore dell'output e successivamente si aggiorna il puntatore out che indica la posizione dove prelevare (allo stesso modo con il modulo circolare) → infine viene decrementato il count!

## Race Conditions

+Risulta possibile però che l'esecuzione in parallelo dei 2 processi possa portare al fatto che l'istruzione di aggiornamento di count sia fatta in modo concorrente → mentre si esegue count++, si esegue anche count-- → in sostanza il problema sta nel fatto che si legge/scrive su una stessa variabile condivisa

+Considerando il compilatore, le due istruzioni di aggiornamento si spezzettano in più istruzioni:

count++ potrebbe essere implementata come

```
register1 = count
register1 = register1 + 1
count = register1
```

count- potrebbe essere implementata come

```
register2 = count
register2 = register2 - 1
count = register2
```

Ipotizziamo quindi di avere una solo CPU che esegue tutte le istruzioni...

Consideriamo questa possibile esecuzione con inizialmente "count = 5":

- S0: produttore esegue **register1 = count** {register1 = 5}
- S1: produttore esegue **register1 = register1 + 1** {register1 = 6}
- S2: consumatore esegue **register2 = count** {register2 = 5}
- S3: consumatore esegue **register2 = register2 - 1** {register2 = 4}
- S4: produttore esegue **count = register1** {count = 6 }
- S5: consumatore esegue **count = register2** {count = 4}

→ dopo S1, arriva una prelazione... finito dunque il time\_slide del thread, si passa ad eseguire l'altro consumatore → si esegue il decremento e rientrà in esecuzione l'altro processo che non fa altro che mettere il contenuto del registro e metterlo nella variabile count → ma proprio dopo il consumatore finisce la sua esecuzione cambiando il valore di count! → per cui non si ottiene il risultato che ci si aspetta dall'incremento di count! (allo stesso modo se voglio fare un decremento e nel mentre si attiva l'incremento)

→ Questo porta ad uno stato **inconsistente** in quanto lo stato finale dopo count++ e count-- dovrebbe essere count=5 indipendentemente dall'ordine!

→ anche nel caso il consumatore fa tutte le sue istruzioni prima che finisca anche il produttore → count alla fine prende il valore 6, ma questo ancora non va bene!

→ il problema è infatti quello di avere 2 processi concorrenti che accedono alla stessa risorsa per leggere un valore e poi modificarlo riscrivendoci sopra...

→ le due attività concorrenti sono quindi dette **sezioni critiche** dei processi → esecuzione concorrente che può portare a malfunzionamento! → può portare infatti... è qualcosa di casuale e il fallimento è totalmente incontrollato → dipende dal tempo... dunque difficile da testare o debuggare

+In generale si parla dunque di *race condition* per i casi in cui vengono eseguite delle operazioni di lettura, modifica e salvataggio di un **dato condiviso** da parte di 2 o più thread (se infatti ogni thread ha una sua copia del dato non avrei problemi... ma quando si parla di stati condivisi fra più processi che lo manipolano si possono avere race conditions...)

+Another example(da un conto bancario di 1000euro):

### Prelievo

- leggo stato conto
- decrementa di 500 €
- salvo stato conto

### Deposito

- leggo stato conto
- incremento di 5000 €
- salvo stato conto

→ se si possono alternare le esecuzioni delle istruzione del prelievo con quelle del deposito (questo può accadere proprio perchè le istruzioni non sono un blocco singolo ma spezzettate) → si può dunque arrivare ad uno stato di inconsistenza alla fine che può essere di 5500euro(quello giusto), oppure solo 500euro, oppure ancora 6000euro (dove è stato buttato via l'aggiornamento fatto nel prelievo)

→ risulta importante dunque considerare questi blocchi di istruzione come indivisibili (spoiler **transazioni** a basi di dati) in modo che le istruzioni vengano serializzate (?) → dunque l'effetto è come se le 2 istruzioni vengano eseguite una dopo l'altra in sequenza! → ottenendo come risultato finale comunque 5500euro!

+è questo un problema a sé stante detto *problema della sezione critica*

→ Valgono alcune proprietà:

- **Mutua esclusione** → più processi non possono essere ognuno nella propria sezione critica → se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può essere anche lui in esecuzione della propria esecuzione critica
- **Progresso** → Se nessun processo è in esecuzione nella sezione critica ed esistono processi che vogliono entrare nella propria sezione critica allora qualcuno ci entrerà! → entrata che non può essere posposta definitivamente (altrimenti avrei stallo)
- **Attesa limitata** → Deve esistere un limite al numero di volte in cui altri processi entrano nella loro sezione critica dopo che un processo ha richiesto di entrare nella sezione critica → proprio per la proprietà precedente → che la richiesta di un altro processo non può essere posposta all'infinito! → deve essere garantito che tutti possano ad un certo punto entrare nella sezione critica.

→ ad esempio nel caso di un processo a più alta priorità riesce ad acquisire la risorsa prima dell'altro... non può impedire all'altro di accedervi!

## 30/04/2021

### Problema della sezione critica

Continuiamo il problema precedente... Vogliamo garantire la caratteristica di consistenza → si fa quindi un esame delle sezione critiche → le quali avranno una prima attività di richiesta di accesso (ovvero di entrata nella sezione critica) → il codice che farà effettivamente parte di questa sezione → e poi una operazione di uscita dalla sezione critica la quale permetterà agli altri di entrare!

→ risulta infatti le proprietà della sezione critica quella precedentemente dette → mutua esclusione - progresso - attesa limitata

→ la prima si capisce abbastanza... la seconda dice che se più di un processo vuole entrare nella propria sezione critica, allora qualcuno entra → non si può postporre indefinitivamente l'accesso alla sezione critica!

→ l'ultima invece ci dice in sostanza che un processo ad alta priorità non può impedire ad altri processi di entrare in maniera indefinita nel tempo

+Si assume che comunque qualsiasi processo venga eseguito a velocità non nulla cioè che non si blocchi... però non sì fa nessuna assunzione sulla velocità relativa degli n processi!  
→ questi possono avere velocità differenti ma comunque deve essere garantita la proprietà di attesa limitata!

## Soluzione di Peterson

è una prima soluzione ma è valida solo per 2 processi → abbiamo 2 processi che vogliono entrare nella loro sezione critica...

+Si assume che i sistemi con i quali sono in esecuzione usano delle istruzioni di LOAD e STORE atomiche → dunque che non possono essere interrotte e questi 2 processi usano 2 variabili condivise → la variabile turn che mi dirà chi avrà il turno ad entrare nella sezione critica e quindi un array o (comunque 2 booleani → uno per ogni processo) che ci dirà se un processo risulta più o meno pronto per entrare nella sua sezione critica

- int **turn**;
- bool **flag[2]**

→ se quindi  $\text{flag}[i] == \text{true}$ , implica che il processo  $P_i$  è pronto ad entrare nella propria sezione critica (dunque aspetta di poter entrare)

|                                                                                                                                                                                                                 |                                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>process <math>P_0</math></b>                                                                                                                                                                                 | <b>process <math>P_1</math></b>                                                                                                                                                                                 |
| do {<br><b>flag[0] = TRUE;</b><br><b>turn = 1;</b><br>while ( <b>flag[1] &amp;&amp; turn == 1</b> )<br>;<br><b>CRITICAL SECTION</b><br><b>flag[0] = FALSE;</b><br><b>NON CRITICAL SECTION</b><br>} while(true); | do {<br><b>flag[1] = TRUE;</b><br><b>turn = 0;</b><br>while ( <b>flag[0] &amp;&amp; turn == 0</b> )<br>;<br><b>CRITICAL SECTION</b><br><b>flag[1] = FALSE;</b><br><b>NON CRITICAL SECTION</b><br>} while(true); |

→ Po inizialmente ha una fase di richiesta/attesa di entrare nella sezione critica → entra quindi in questa sezione, e una volta uscito si rilascia, la sezione critica e infine c'è il codice che non fà parte di questa sezione → si suppone che tutto si ripeta iterativamente

→ il processo  $P_1$  sarà fatto in modo analogo soltanto opera su variabili e mette valori leggermente diversi →  $P_1$  usa infatti il flag 1 per dire che il processo è pronto per entrare nella sezione critica → quando invece uscirà imposterà il flag a falso, per dire appunto che non ha più bisogno di essere nella soluzione critica (tramite il flag)

→ la variabile turn dirà di chi sarà il turno (quando entrambi sono pronti ad entrare) → dunque nel caso  $\text{turn} = 0$  indica che  $P_0$  è pronto ad entrare dunque  $P_1$  lo fa passare! → si attende dunque che o cambi il turno oppure se il l'altro processo non vuole più entrare nella sezione critica, mentre il processo "ospitante" aspetta (ciclo while)

→ notare che l'impostazione di  $\text{flag}[0] = \text{false}$  o quello flag 1, sblocca quell'altro processo!

+Adesso ci chiediamo... saranno rispettate le proprietà richieste??

• **Mutua esclusione:** se entrambi fossero nella sezione critica si avrebbe che  $(\text{flag}[1]==\text{FALSE} \ || \ \text{turn}==0) \ && (\text{flag}[0]==\text{FALSE} \ || \ \text{turn}==1)$

Ma  $\text{flag}[0]==\text{flag}[1]==\text{TRUE}$  (perché le imposto io all'inizio → prima istruzione) quindi deve valere  $\text{turn}==0 \ && \ \text{turn}==1$  (assurdo) → o uno o l'altra! (in particolare se non è uno è sicuramente 0 → non 2 o altri valori) → questo ci garantisce di avere mutua esclusione!

- **Progresso:** se P0 e P1 vogliono entrare sicuramente uno dei due entra (dipende dal valore di turn) → a seconda di quale processo viene schedulato prima turn sarà 1 o 0 → uno entra
  - **Attesa limitata:** dopo che P0 è entrato e P1 è in attesa, se P0 richiede nuovamente di entrare → ad esempio nel caso in cui P0 abbia più priorità, anche nel caso lo scheduling lo avvantaggia viene comunque prima sbloccato P1! → infatti settando turn = 1, nel processo P1 questo rende falso il while dunque entra nella parte critica! (mentre P0 aspetterà comunque anche con priorità maggiore)
- viceversa P0 non può comunque rientrare subito, infatti soddisfa appieno il while, dunque aspetta e lascia passare P1 → dunque P1 si dice avere attesa limitata! → non rimane in attesa per troppo tempo!

→ algoritmo teorico, valido solo per 2 processi...

### Algoritmo del fornaio(o del pollo :))

Fornisce una soluzione nel caso di N processi!

→ Simile a quando andiamo a prendere le patatine e il pollo (o il pane), si prende il nostro numerino che serve per dare accesso alla risorsa che il singolo fornaio che mi deve servire  
→ dunque preso il numerino ognuno aspetta che venga chiamato il proprio numero!

+Si usano 2 vettori condivisi tra gli n processi che sono il vettore dei choosing (è un booleano) → 1 per ogni processo che indicherà quando è vero che il processo è in fase di scelta del suo numero (vettore detto *choosing[i]*)  
→ l'altro sarà *number[i]* che indica il numero preso dal processo i-esimo, per entrare nella sezione critica e zero mi indicherà che il processo non vuole entrare nella sezione critica → il numero per essere valido deve essere un numero maggiore di zero! (altrimenti se zero significa che non vuole entrare nella sezione critica)

+L'algoritmo per l'i-esimo processo sarà quindi

```
process P(i) {
 do {
 choosing[i] = true;
 number[i] = 1 + max(number[0], ..., number[N-1]);
 choosing[i] = false;
 for(j=0; j<N; j++) {
 while(choosing[j]) {}
 while(number[j]!=0 && (number[j],j) < (number[i],i)) {}
 }
 CRITICAL SECTION
 number[i] = 0;
 NON CRITICAL SECTION
 } while(true);
}
```

→ tutti i vari processi eseguono quindi le stesse istruzioni...  
→ la prima fase sarà quella della scelta del numero → impostando l'elemento i-esimo del vettore choosing a vero per dire che il processo è nella fase di scegliere il numero → il numero viene dunque preso come 1+ il max fra i numeri presenti nel vettore dei numeri appunto → si cerca il massimo fra tutti i processi e se tutti risultano a zero significa che

nessuno sta richiedendo di entrare nella sezione critica → il processo i-esimo sarà il numero 1 e quindi il primo ad entrare!

→ notare che la funzione di max è una funzione che opera su diversi processi in modo contemporaneo → per cui posso avere una situazione nella quale più processi prendono lo stesso numero! → situazione che dovremmo gestire in qualche modo...

→ una volta scelto il numero si itera su tutti i processi (anche se stesso) → guarda se il processo i-esimo si trova nella fase di scelta del numero → se un processo è in questa fase allora aspetta che quel processo abbia scelto un determinato numero (verifica la condizione anche per se stesso → questo sarà sicuramente falso → dunque non aspetta)

→ si passa quindi al successivo while dove si controllo se il valore del numero è diverso da zero → se così non fosse vuol dire che il processo ha già fatto quello che doveva fare ed è quindi fuori dalla sezione critica → se quindi è zero si passa al processo successivo

→ viceversa si guarda se il numero preso dal processo j-esimo è più piccolo del suo

→ come anticipato c'è però la possibilità che 2 o più processi abbiano preso lo stesso numero allora il confronto viene fatto sulla coppia numero scelto dal processo nella fase iniziale e il numero del processo → un identificatore in un qualche modo → e come scritto nel quadrato in blu(vedi figura) nel caso si abbia stesso numero si confronta il secondo numero → e in questo caso passerà il numero a identificatore (o indice) minore → ovvero il processo a più alta priorità! → il processo 0 passerà prima degli altri processi (non si considera il caso

j = i, ovvero se stesso che comunque porterebbe ad un false)

→ arrivato in fondo alla scansione → se ci arriva, significa che lui ha il numero più piccolo rispetto a tutti gli altri → che eventualmente chiedono l'accesso e dunque può entrare nella sezione critica!

→ fatto quello che deve fare, pone il suo numero preso a zero, in modo da sbloccare altri processi che eventualmente vogliono entrare nella sezione critica!

+Anche sta volta mi chiedo... sono rispettate le proprietà?

■ **Mutua esclusione:** al termine del ciclo  $(\text{number}[i], i) < (\text{number}[j], j)$  con  $j=0..n-1$  e  $j \neq i$ , se due processi x e y fossero entrambi nella sezione critica si avrebbe  $(\text{number}[x], x) < (\text{number}[y], y)$  e si avrebbe contemporaneamente la proprietà opposta anche per l'altro processo ovvero  $(\text{number}[y], y) < (\text{number}[x], x)$  → assurdo! (valido per  $x = y$  solo, ma io li suppongo diversi!) → un solo processo può essere nella sezione critica

■ **Progresso:** se nessuno è nella sezione critica e più processi vogliono entrare sicuramente uno verrà scelto → in particolare quello che avrà la coppia  $(\text{number}[i], i)$  minore!

■ **Attesa limitata:** i valori di  $\text{number}[]$  aumentano sempre, se un processo è in attesa prima o poi  $(\text{number}[i], i)$  sarà il più piccolo tra quelli presenti (fra quelli che stanno chiedendo di entrare) → dunque prima o poi entrerà! → se un processo inizialmente a number minore finisce e poi chiede di rientrare avrà poi un number maggiore!

## Hardware per la sincronizzazione

In realtà questi algoritmi ormai non vengono più utilizzati e in realtà si ha che i sistemi/processori forniscono un supporto per l'attività della sincronizzazione...

→ le possibilità sono 2:

- Nel caso dei *monoprocessori*, si ha la possibilità di disabilitare le interruzioni → in modo da disabilitare le attività di prelazione! → per cui non ci può essere un processo che entra durante l'esecuzione della sezione critica... → si potrebbe fare anche nei multiprocessori, ma risulta un po' inefficiente perché comunque blocca l'esecuzione su tutti i processori perchè magari solo un processo ha deciso che non entra nella sezione critica → gli altri invece potrebbe voler fare altre cose non necessariamente collegate a quello che sta facendo il processo in esecuzione su quel processore!

→ nelle macchine moderne abbiamo in realtà delle istruzioni **atomiche** speciali → ovvero istruzioni che non sono interrompibili e che permettono di gestire questa sincronizzazione... Anche qui 2 tipi (prima boh ce ne era solo 1) → che controllano il valore e poi lo impostano oppure scambiano il contenuto di 2 parole di memoria

### TestAndSet

+Ad esempio l'istruzione TestAndSet → legge e modifica senza essere interrotto

```
bool TestAndSet (bool *target)
{
 bool rv = *target;
 *target = TRUE;
 return rv: 3
```

→ una operazione eseguita dal processore come indivisibile che legge il valore che ha una variabile → la mette a vero e quindi di ritorna il valore che aveva precedentemente  
 → la caratteristica è questa operazione NON può essere interrotta ed è quindi eseguita sul processore senza poter permettere un'interruzione tra l'esecuzione delle 2 attività! (ovvero di lettura e scrittura → sicuramente se eseguiamo l'attività di lettura viene eseguito immediatamente anche quella di scrittura!)

+Come utilizzare dunque questa istruzione per gestire il caso di sezione critica? → uso della variabile lock

```
do {
 while (TestAndSet (&lock))
 ; /* do nothing */

 //critical section

 lock = false;

 //non critical section

} while (true);
```

→ si esegue nell'while l'istruzione testAndSet di una variabile condivisa che contiene un valore booleano → questo sarà vero per dire che la sezione critica è bloccata (ovvero è in

uso) → prima del while è quindi false, mentre testAndSet legge il valore della variabile **lock** e lo imposta a vero → fin tanto che dunque questo è vero lui itera → in particola prende e imposta a vera la variabile lock

→ solo quindi quello che riesce a trovare come risultato false riesce ad entrare! → gli altri invece la trovano già bloccata proprio per il fatto che l'attività dell'istruzione testAndSet non risulta interrompibile!

→ come al solito nel momento in cui ha finito di usare la sezione critica, si imposta il lock a false e lascierà a qualcun altro la possibilità di entrare nella sezione critica... quello che riesce a beccare la variabile lock a false riuscirà ad entrare nella sezione critica, e solo uno alla volta riesce ad entrare!

+In questo schema qui in realtà si può avere però una situazione in cui abbiamo un attesa indefinita → ad esempio nel caso di processi che hanno più priorità rispetto ad altri e che addirittura hanno una sezione critica non nulla... questo continuamente ha accesso in modo indefinito alla sezione critica lasciando gli altri a bloccato! → attesa limitata non garantita! (le prime 2 proprietà sì)

## Swap

+Un'altra possibilità è quella di usare l'operazione di scambio → swap in memoria → un'attività che prevede lo scambio di 2 parole di memoria diverse (astrazione) → qualcosa fornita dal processore che è equivalente all'attività di scambio

```
void Swap (bool *a, bool *b)
```

```
{
```

```
 bool temp = *a;
 *a = *b;
 *b = temp; }
```

→ breve recap su come opera lo swap → leggo il ptr a e lo metto in una variabile temporanea → poi salvo in a il valore di b e infine salvo in b il valore precedente → anche questa è un'attività fatta che non può essere interrotta → come fosse una singola istruzione → come si utilizza quindi questa funzione?

```
do {
 key = true;
 while (key == true)
 Swap (&lock, &key);
 //critical section
 lock = false;
 //non critical section
} while (true);
```

→ si utilizza di nuovo la variabile lock → che è condivisa e inizializzata a false, ma in più ogni processo ha una variabile locale key → che contiene ciò che vuole fare il processo → la fase di richiesta di entrata nella sezione critica, mette il key a true → dunque fino a che key risulta vero, prova a scambiare il valore di lock con il valore di key → per cui supponendo che l'attività non si possa interrompere se lock ha valore falso key diventa falso

e dunque si esce dal ciclo e si blocca l'accesso alla sezione critica, viversa se lock era già posto true da un altro processo allora anche key rimane true → e si continua il ciclo  
 → nel momento in cui key riesce a trovare lock a false → allora si scambiano il valore e solo uno degli n processi che stanno tentando di accedere alla sezione critica riuscirà a prendere il valore di lock a false e quindi a poter entrare nella propria sezione!  
 → come al solito terminata la sezione critica, si imposta lock a false dando possibilità ad altri processi di poter entrare  
 → anche questo sistema garantisce la mutua esclusione e anche attività di progresso → in quanto comunque un processo troverà lock a false → almeno uno riesce ad entrare...  
 → rimane però il problema dell'attesa limitata → potrebbe infatti esserci un processo che prenda l'accesso alla sezione critica impedendo per lungo tempo agli altri l'accesso!

### Soluzione con attesa limitata

In entrambi i casi precedenti le soluzioni hardware viste garantiscono la mutua esclusione e progresso ma non l'attesa limitata!

→ si usa quindi la seguente soluzione → per il Processo Pi

```
do {
 waiting[i] = true;
 key = true;
 while(waiting[i] && key)
 key = TestAndSet(&lock);
 //CRITICAL SECTION
 j = (i+1) % n;
 while(j!=i && !waiting[j])
 j = (j+1)%n;
 if(j==i)
 lock = false;
 else
 waiting[j] = false;
 //NON CRITICAL SECTION
}while(true);
```

waiting[i] indica se il processo Pi è in attesa di entrare in sezione critica (inizializzato a false)

Cerca il prossimo processo che è in attesa sul lock, seguendo ordine i+1..n-1,0,1,... i e se lo trova lo sblocca. Questo garantisce attesa massima di n-1 rilasci

→ si usa l'array di waiting dove la posizione i-esima dell'array, indica se il processo i è in attesa di entrare nella sezione critica

→ si usa di nuovo la variabile key impostandola a true, e fintanto che waiting[i] e key sono true (indicando che il processo si trovi in uno stato di attesa) → si fa il testAndSet del lock → richiedendo di accedere alla sezione critica (si poteva fare anche lo swap)

→ nel momento in cui il processo “è fortunato” e trova il lock a false anche la key assume valore falso e quindi il processo riesce ad entrare nella sezione critica

→ a questo punto il rilascio della sezione critica diventa un po' più difficile... si guarda se ci sono altri processi che sono in attesa → preso quindi un processo che ha già completato la sua sezione critica → va quindi a guardare gli altri processi partendo da quello successivo a lui stesso → trova quindi il j-esimo processo in attesa (potrebbe anche accadere che scansionando gli altri processi fino a lui nessun altro risulti in attesa...)

→ se quindi ne trova 1 che è in attesa oppure è riuscito a tornare a se stesso → in quest'ultimo caso vuol dire che nessuno altro processo è in attesa e quindi lui può lasciare il lock → ponendolo a falso...

→ viceversa se ne trova 1 in attesa allora mette lo waiting[j] a falso → sbloccandolo dalla fase di attesa, ma mantenendo il key a true → comunque il processo j-esimo riesce ad entrare nella sezione critica!

→ questa attività permette di garantire un'attesa massima di n-1 rilasci → infatti prima o poi chi cerca di entrare sarà sbloccato da qualcuno! → un processo non può accedere continuamente alla sezione critica non rilasciando nessun altro! → questo garantisce un attesa limitata!

+Le soluzioni viste fino a questo momento sono *CPU intensive* → ovvero usano la CPU per controllare ripetutamente valori di uno o più variabili per verificare se lo stato sia cambiato o meno! → questo tipo di soluzioni sono dette *busy waiting* o ad *attesa attiva* → ovvero la CPU è usata per controllare lo stato e usare la CPU per questo tipo di controlli risulta un po' inutile... potrebbe infatti essere usata per eseguire altri programmi!

→ supponendo infatti di avere una sezione critica di un processo molto lunga...tutti gli altri processi in attesa sono lì costantemente ad usare la CPU per fare questo tipo di controllo!

+Per questo motivo i sistemi operativi forniscono delle *primitive di sistema* → per realizzare la sezione critica → dunque per gestire l'accesso al codice che deve essere eseguito in sezione critica...

→ in genere queste primitive fornite dal sistema sono dette *mutex* (da **mutal exclusion**) → le quali servono a garantire l'esecuzione di parti del codice in modalità di mutua esclusione → ovvero solo uno fra più contendenti riesce ad accedere a qualcosa!

→ in particolare sono (in genere) implementati senza ricorrere all'attesa attiva → sistema operativo che fa in modo di non usare l'attesa attiva → si mette infatti il processo che è in attesa in una lista di attesa → dunque gestito come se fosse un device che aspetta l'I/O da parte di qualcosa → solo che stavolta aspettano un mutex!

→ quando poi il mutex sarà rilasciato, potrà essere ripresa l'esecuzione del processo in attesa! (approccio che risulta decisamente conveniente nel caso di sezioni critiche abbastanza lunghe!)

+I sistemi operativi forniscono inoltre spesso anche gli *spinlock* → in cui invece viene usata l'attesa attiva → in alcuni casi infatti è più vantaggioso usare questo, in quanto il costo del cambio di contesto, risulta più costoso rispetto al costo del codice nella sezione critica!

→ ad esempio nel caso di sezioni critiche con una sola istruzione → in quel caso il costo per mettere il processo in coda di attesa / fare il context switch su un altro processo lasciando la CPU libera di fare altro / infine la parte per l'esecuzione delle istruzioni critiche (molto breve rispetto alle altre) → per cui risulta più conveniente fare attesa attiva... (magari anche gli altri processi fanno pochi controlli...)

→ quando il tempo di esecuzione della sezione critica è basso conviene comunque usare l'attesa attiva → impiegherà infatti troppo tempo per contenxt switch

## Semafori

Astrazione fornita dai sistemi operativi → semafori in particolare che sono un modo per implementare i *mutex* !

→ è quindi uno strumento di sincronizzazione dove un semaforo è associato ad una variabile intera → per questo semaforo abbiamo 2 operazioni standard che modificano lo **stato** del semaforo → *wait()* e *signal()*

→ un semaforo fa quindi la wait e la signal che sono definite in questo modo:

```

• wait (S) { • signal (S) {
 while (S <= 0) S++;
 ; // no-op }
 S--;
}

```

- la wait dunque appunto rimane in attesa fintanto che il valore del semaforo (appunto un intero) è negativo → nel caso in cui invece lo trova con un valore positivo ne decrementa il valore → vicensa la signal semplicemente ne incrementa il valore
- entrambe le operazioni sono indivisibili → nel senso che se si riesce a leggere un valore positivo (ad esempio da zero passa a 1 tramite una signal che rilascia il semaforo) → se riesce a leggere 1 sarà l'operazione di wait stessa a portarlo a zero!
- ipotizziamo ad esempio di avere tutti i processi che sono in attesa sullo stesso semaforo, poi abbiamo un altro processo che fà la signal... questo riesce svegliare solo uno di quelli che sono in attesa → solo uno di quello che sono in attesa riuscirà a trovare libero il semaforo e quindi a poter entrare nella sezione critica!

Come si usano sti semafori?

Abbiamo i semafori *contatori* → nei quali il valore intero non viene vincolato...

→ poi sono presenti i semafori *binari* → nei quali il valore intero può essere invece 0 o 1 → dunque più semplice da implementare → infine sono conosciuti anche come *mutex locks* anche se possiamo implementare un semaforo binario da uno contatore!

→ come si fornisce la mutua esclusione usando i semafori?

```

Semaforo mutex=1;

wait(mutex);

Critical Section

signal(mutex);

```

→ poniamo il semaforo a 1 → chiamiamo la wait sul mutex stesso → eseguiamo la sezione critica e infine chiamiamo la signal sul mutex → quindi quando il processo fà la wait riuscirà a mettere la mutex a zero e nel caso di più processi che fanno la wait sullo stesso mutex → solo uno riuscirà a decrementare da 1 a 0... gli altri rimangono in attesa!

## Uso dei semafori

I semafori contatore si possono usare per regolare l'accesso ad un numero n di risorse che vengono utilizzate singolarmente → in questo caso si inizializza il valore del semaforo a n che dice appunto quante risorse sono a disposizione → nel momento in cui abbiamo bisogno di accedere ad una risorsa, prima di prelevarla si farà la wait

→ per cui se le risorse sono terminate... ovvero se S = 0 → allora la chiamata a wait fà entrare in uno stato di attesa

→ nel momento in cui poi il processo ha terminato di usare la risorsa che aveva precedentemente acquisito facendo la wait → allora usa la signal per segnalare appunto la fine dell'uso di quella risorsa e quindi si incrementa il valore di quel semaforo → e quindi se

qualche thread era portato in attesa sulla wait → perchè ad esempio le n risorse erano terminate → allora la signal sbloccherà per thread/processo che era in attesa  
→ se però più di un processo era in attesa di una risorsa → ad esempio abbiamo 100 processi che devono accedere a 4 risorse → 4 processi riusciranno a prenderle mentre il resto aspetta! → nel momento in cui quelli che hanno avuto accesso alla risorsa terminano di usarla → daranno la signal facendo “svegliare” uno di quei processi che era in attesa con il semaforo a zero → permettendo ad altri processi di accedere alla risorsa!

+Un'altra possibilità dell'uso dei semafori è quella di sincronizzare l'uso di parti di codice tra 2 thread → si ipotizza di avere 2 thread:

```
Thread1: Thread2:
...
...
S1;
signal(synch); -----> wait(synch);
...
S2;
...
...
```

→ vogliamo garantire che il codice S2 sia eseguito sempre dopo il codice S1 → supponendo ad esempio che il risultato del codice S1 venga utilizzato quindi dal codice S2  
→ decido dunque di usare un semaforo binario detto *synch* inizializzato a zero  
→ ipotizziamo quindi che venga eseguito prima il thread 2 → l'esecuzione si ferma quindi al wait, trovando il *synch* a zero! → per cui si aspetta! → parallelamente(o dopo?) viene eseguito l'altro thread → il quale esegue tutto il codice di S1 e quando ha finito fà la signal sul semaforo che quindi lo incrementa portandolo a 1 → questo evento sblocca l'altro thread il quale esce dall'attesa e poi eseguirà S2  
→ cosa simile accade anche se viene prima eseguito il thread1 → infatti eseguito tutto S1, con la signal porta *synch* a 1 e continua la sua esecuzione → quindi il thread2 va in esecuzione → fà la wait su *synch* ma essendo già 1 non c'è bisogno di aspettare per cui esegue S2!

## Implementazione dei semafori

→ si deve garantire che 2 processi non possono eseguire *wait()* e *signal()* sullo stesso semaforo e allo stesso istante di tempo!  
→ nell'implementazione vista finora abbiamo visto che comunque la *wait* esegue un attesa attiva → con il *while* sul valore del semaforo... che occupa la CPU inutilmente durante l'attesa che la sezione critica sia libera → siccome le applicazioni possono passare molto tempo nella sezione critica questa non è una buona soluzione...

+Si usa dunque una soluzione che sfrutta delle code → ad ogni semaforo viene associato una coda di attesa → abbiamo quindi a disposizione 2 operazioni associate per i semafori:

- **block** – inserisce il processo che invoca l'operazione in una coda di *attesa* e il processo non entrerà più nella coda dei processi *ready* e quindi non sarà più eseguito!
- **wakeup** – rimuove uno dei processi nella coda di *attesa* e lo mette nella coda dei processi *ready*

→ dunque il semaforo non viene definito come un solo valore intero, ma con un valore intero più una **struttura** che sarà una lista dei processi che sono in attesa sul semaforo!

```

typedef struct {
 int value;
 struct process* list;
} semaphore;

```

+Come verranno quindi implementate le funzioni di wait e signal in questo caso?

```

wait(semaphore* S) {
 S->value--;
 if(S->value < 0) {
 aggiunge il processo alla coda S->list;
 block(); } }

```

→ si diminuisce inizialmente il valore del semaforo e successivamente si verifica se il valore sia negativo → in quel caso il processo viene aggiunto alla coda list → e si chiama la funzione block() ! → il processo si blocca in quel punto e resta in attesa che qualche altro processo lo svegli! (ovvero qualcuno chiami la funzione wakeup())!

+Per la signal invece:

```

signal(semaphore* S) {
 S->value++;
 if (S->value <= 0) {
 rimuove un processo P dalla coda S->list;
 wakeup(P);
 } }

```

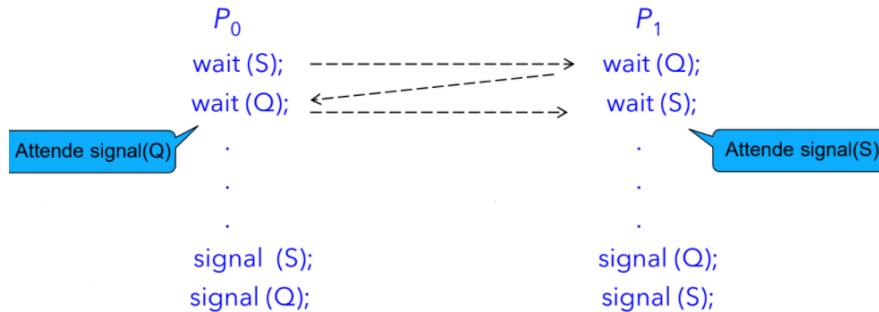
→ prima cosa si incrementa il valore del semaforo → nel caso questo valore sia negativo (o uguale a zero) si rimuove il processo dalla coda S->list e si chiama il wakeup(P) → svegliando il processo che era in attesa che era sulla wait!

→ entrambe le operazioni devono essere seguite in una sezione critica o disabilitando le interruzione nel caso di sistema monoprocesso oppure usando uno spinlock nel caso multiprocessore → in quanto l'attività di incremento/decremento se fatte in modo concorrente possono generare una situazione di inconsistenza!

+Se il valore del semaforo risulta negativo ci dice appunto il numero dei processi che sono in attesa! → che sono bloccati su quel semaforo... supponendo ad esempio di aver raggiunto il valore -3 per il semaforo, che appunto indica di avere 3 processi in attesa (se invece è zero non ce ne sono) → si chiama quindi la signal → S sarà incrementato di 1, quindi passa a -2 → si soddisfa la condizione del semaforo per cui ho ancora dei processi in attesa... si rimuove comunque uno dei processi in attesa chiamando la funzione di wakeup sul processo P → e così via fino a che non ci sono dei processi in attesa!

## Stallo e attesa indefinita

Due processi possono aspettare indefinitivamente un evento che ad esempio è causato da uno dei processi in attesa → in particolare abbiamo 2 processi che si bloccano a vicenda:



→ supponiamo che siano P e Q due semafori inizializzati a 1 → abbiamo 2 processi P0 e P1 dove il primo fa la wait su S e poi su Q e alla fine chiama la signal su S e infine su Q → allo stesso modo ma in maniera opposta agisce P1 → i 2 processi dunque agiscono in questo modo → si parte con P0 che chiama wait(S) → poi questo viene prelazionato → l'esecuzione passa a P1 che fà wait(Q) → ed essendo i semafori inizializzati a 1 passano entrambi a zero → per cui quando nel processo P0 si chiama wait(Q) lui si blocca e rimane in attesa → si passa quindi al processo P1 che anche lui entra in attesa facendo wait(S) → il problema è che chi può chiamare la signal su P e Q sono i 2 processi in attesa! → e non potendo raggiungere quel pezzo di codice i 2 processi rimarranno bloccati → per cui si parlerà di situazione di **stallo**

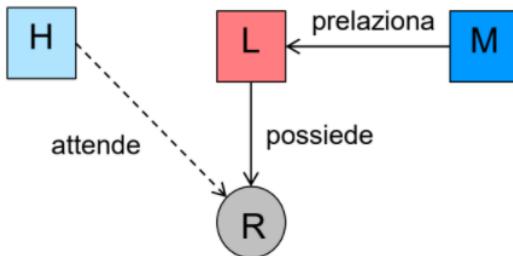
→ non sempre si entrerà in questa situazione → ad esempio se il processo P0 esegue sia la wait(Q) che wait(S) non entrando in prelazione... allora non si blocca ma pone solo i semafori entrambi a 0! → a quel punto gli altri processi come P1 si bloccheranno dopo ad esempio wait(S) ma non avrò una situazione di stallo! (P0 infatti riesce a chiamare le 2 signal e quindi sbloccare il processo P1)

→ in sostanza il comportamento dei 2 processi dipende fortemente dallo scheduling di questi → il quale però risulta un evento sulla quale il processo non ha nessun controllo... → si parla quindi di **attività casuale** → proprio perchè l'attività dipende dallo scheduling, dal tempo, da come si gestisce l'esecuzione dell'insieme dei processi → per cui lo stallo risulta un evento casuale che può accadere o meno e noi non lo sappiamo...  
→ il che rende il debug di questo tipo di applicazioni molto complicato.

+L'altra possibilità è quella di **attesa indefinita** in cui un processo può non essere mai rimosso dalla coda di un semaforo dove è sospeso → questo può accadere nel caso di code gestite in un ordinamento diverso → ad esempio LIFO (rispetto a FIFO)

## Inversione di priorità

Supponiamo di avere 3 processi → uno a priorità bassa, l'altro a media e infine a alta, inoltre abbiamo un risorsa → a cui si accede tramite semaforo



→ abbiamo che il nostro processo a bassa priorità che in quel momento sta usando la risorsa → la quale risulta quindi bloccata per lui...

Supponiamo quindi che ad un certo punto il processo a più alta priorità, vuole usare la risorsa... siccome l'accesso è a mutua esclusione, lui dovrà comunque aspettare...

Il problema però sta nel fatto che arrivi il processo a priorità media ( $H > M > L$ ), il quale prelaziona il processo a priorità più bassa → dunque passa in esecuzione M che sta bloccando L e indirettamente sta bloccando H! → infatti L essendo bloccato non riesce a rilasciare la risorsa, per cui H sta aspettando...

per cui abbiamo la situazione in cui un processo a più bassa priorità sta bloccando l'esecuzione di uno a più alta priorità!

+Come si risolve questo problema?

Una possibilità (anche se non la migliore) è limitare le priorità solo a 2 livelli di priorità...

→ La soluzione più generale è invece quella di **aumentare** la priorità di L portando ad essere uguale a quella di H, ma solo durante l'attesa di H → in modo che M non possa entrare nel mezzo! → portando infatti L a priorità H → M non è in grado di prelazionare L! (per priorità minore)

→ per cui L arriverà in fondo al suo programma → rilascerà la risorsa e H potrà agire liberamente → per cui M non è in grado di bloccare H

→ questa soluzione è detta *protocollo di eredità della priorità*

→ nel momento in cui il processo a priorità alzata, rilascia la risorsa → allora la sua priorità ritorna ad essere bassa!

+ “Esperimenti in java” (da riscrivere → 1:36:00)

→ ricordarsi del costrutto *volatile* → utile per far evitare situazioni in cui il processore cercando di ottimizzare il programma non fà il controllo attivo di un processo → senza il volatile non si va a guardare effettivamente se il valore della variabile è cambiato o meno... senza volatile non funziona nulla!

+occhio però che se vogliamo costruire un array di elementi volatile non basta aggiungere volatile alla definizione (in quel caso volatile saranno i riferimenti) ma dobbiamo usare una particolare struttura dati detta *automi per integers/array?* (oppure semplicemente usiamo le variabili singole)

# 5/05/2021

## Semafori in java

→ In Java si possono usare oggetti della classe **Semaphore** (che fa parte del package `java.util.concurrent`) che implementa i semafori contatori

→ la classe fornisce dei metodi di base che sono: `acquire()` e `release()` e sono gli analoghi di `wait()` e `signal()` → acquire permette di decrementare il valore del semaforo e release permette di incrementarlo → come per la wait, il metodo acquire farà entrare in attesa se il semaforo è zero, mentre la release non entra mai in attesa

+Il semaforo può essere *fair* o *unfair* → se fair le chiamate in attesa su acquire vengono accodate su una coda con ordine FIFO, se unfair **non** garantiscono ordinamento → ma ha il vantaggio di essere più veloci

+Il costruttore della classe semaforo vuole due parametri → il valore iniziale del semaforo e un valore booleano per indicare se fair/unfair, se viene omesso il semaforo è UNFAIR  
Come si usa?

```
Semaphore mutex=new Semaphore(1,true); //fair
...
mutex.acquire();
count++; //sezione critica
mutex.release();
```

→ mutex è un nuovo semaforo inizializzato a 1 ed è fair in quanto messo a true → per cui userà una coda FIFO per accodare le richieste in entrata → successivamente ci saranno le chiamate `mutex.acquire()` per acquisire il valore del semaforo → dove in questo caso poichè mutex scende a zero nessun altro potrà quindi usare il valore del semaforo! (e terrà in blocco se un altro thread chiede l'acquire!)

→ viene quindi eseguito il codice che appartiene alla sezione critica e una volta finito si chiama il `mutex.release` che permette di incrementare il semaforo e quindi porterà ad incrementare l'accesso alla sezione critica di un altro thread che è in attesa sullo stesso semaforo → potrei avere più thread separati che però accedono a semaforo diversi e questi sono indipendenti fra loro (a seconda se il semaforo sia condiviso o meno)

→ l'errore comune da evitare è quello di mettere in una classe un semaforo associato ad una classe e poi se ne fà un altro associato ad un'altra classe → dunque si creano 2 istanze diverse → che risultano difficile da sincronizzare! → infatti questi vivono in modo separato... è quindi fondamentale che 2 o + thread condividano uno stesso semaforo e non che ognuno si crei un proprio semaforo (perdendo anche l'utilità del semaforo → nel caso di ogni semaforo associato ad un thread)

+Altri metodi che si possono usare su un oggetto di tipo semaforo... si possono usare l'`acquire/release` (parametrico) → `acquire(n)` e `release(n)` che permettono di acquisire un numero arbitrario di *permessi* → nel senso di cose da poter chiedere ad un semaforo → dato un valore lo posso dec/inc di n (che di solito n è 1)

→ nel caso però non siano tutti i valori per cui è richiesto un acquire questo si blocca! → ad esempio se chiedo `acquire(2)` → se il semaforo contiene un valore 1 → banalmente l'acquire di 2 si blocca → in quanto non ci sono a disposizione 2 valori da poter decrementare evitando di arrivare ad un valore negativo → Nel caso di `acquire(n)` aspetterà che il valore del semaforo sia tale da permettere di sottrarre il valore indicato!

+Quale sarà quindi l'uso tipico di un semaforo?

```

Semaphore mutex = new Semaphore(1);
mutex.acquire();
try{
 ... sezione critica ...
} finally {
 mutex.release();

}

```

il semaforo rilasciato

→ creo l'istanza del semaforo dandogli un valore iniziale → chiedo quindi il mutex.acquire → e a seguire metto un blocco try - finally sulla sezione critica

→ perchè si usa finally? → questo per garantire che nel caso in cui nella sezione critica c'è la possibilità di lanciare un'eccezione → e nel caso venga lanciata comunque il mutex venga rilasciato! (ora se la sezione critica ha solo l'istruzione count++ che molto difficilmente può lanciare un'eccezione → dunque non sempre va usato try-finally)

→ risulta però fondamentale in quanto se una risorsa non viene rilasciata allora può tenere bloccata il resto dei thread che usano lo stesso semaforo! (finally nel caso non gestiamo l'eccezione con un blocco catch!)

+ *Tryacquire()* è un altro metodo che in realtà fa l'acquire ma senza aspettare → ritorna un booleano e se l'acquire che si sta tentando di fare non ha successo (ovvero il semaforo è zero → e dunque si otterrebbe un numero negativo) → allora si ottiene false (non mettendo in attesa) → viceversa se ha successo torna true!

→ anche questo metodo agisce in modo atomico → ovvero nel mezzo non può essere bloccato questo thread! → per cui se due thread provano a fare l'acquire contemporaneamente solo uno dei 2 ritorna true (riuscendo a fare l'acquire) e l'altro false!

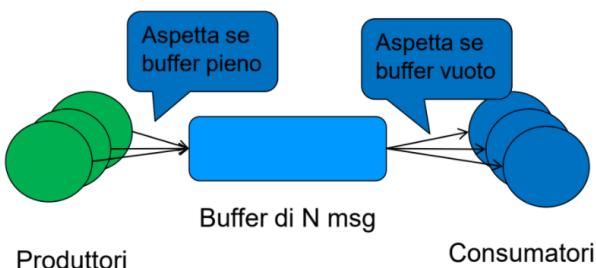
+ Nel caso di semaforo unfair è possibile avere attesa indefinita (anche se abbastanza improbabile)

+ Sono disponibili anche altri classi all'interno del package java.concurrent... che sono utili per gestire dei problemi di sincronizzazione:

- **ReentrantLock** per gestire mutua esclusione e variabili condizione
- **ReentrantReadWriteLock** per gestire lock di lettura/scrittura(casi particolari) → differenzia accesso in lettura e accesso in scrittura...

## Problemi di sincronizzazione

Produttori e consumatori con memoria limitata



Abbiamo più thread **produttori** che vogliono inserire dei dati/messaggi all'interno di un buffer il quale però ne può contenere al massimo  $n!$  →  $n$  messaggi che in particolare formano una coda → inoltre abbiamo altri thread **consumatori** che invece appunto vogliono consumare i dati da questa coda

→ ricordo il vincolo che il buffer è limitato → ci possono stare al massimo  $n$  messaggi  
→ i produttori dunque dovranno aspettare se il buffer risulta pieno! → questo accade spesso nel caso i produttori sono tanti o molto veloci → che riempiono tutto il buffer → dunque quando questo sarà pieno per fare un nuovo inserimento dovranno aspettare che i consumatori → consumino per liberare lo spazio da poter usufruire  
→ allo stesso modo, ma in modo opposto i consumatori dovranno aspettare nel caso il buffer sia vuoto! → nel caso ad esempio i produttori sono lenti il buffer sarà spesso vuoto per cui i consumatori devono aspettare

+Come si implementa il tutto usando i semafori?

→ abbiamo un buffer di  $n$  elementi (ad esempio una coda)  
→ un semaforo **mutex** → che serve a gestire la mutua esclusione per accedere al buffer  
→ un semaforo **piene** (semaforo contatore) → che serve a sapere quanti messaggi all'interno del buffer sono riempiti → cioè quanti valori ci sono all'interno da dover essere consumati??  
→ inizialmente il semaforo sarà inizializzato a zero per dire appunto che all'inizio non c'è nessun messaggio all'interno e quindi nessun valore da prelevare  
→ un semaforo **vuote** → che opera all'opposto → è inizializzato a  $n$  → numero di messaggi che ci possono essere all'interno del buffer e quindi indica in realtà quante posizioni del buffer sono vuote! (all'inizio sono tutte vuote e vuote vale  $n!$ )

+Esempio di processo/thread che produce i dati:

```
do {
 // produce un elemento
 wait(vuote); // decrementa numero posizioni vuote e aspetta se <= 0
 wait(mutex); // evita che produttore e consumatore accedano
 // contemporaneamente al buffer
 // aggiunge l'elemento al buffer
 signal(mutex);
 signal(piene); // incrementa numero posizioni piene ed eventualmente
 // sveglia processo consumatore in attesa
} while (true);
```

→ si fa inizialmente una wait su vuote → appunto nel caso non ci siano posti liberi si aspetta, all'inizio generalmente non si aspetta appunto perché vuote è inizializzato a  $n!$   
→ fatto questo si usa il mutex per bloccare l'esecuzione del buffer → solo infatti un thread alla volta può modificare il buffer → questo regola l'accesso alla struttura dati che contiene i messaggi/dati della nostra lista → mutex per garantire l'uso consistente della coda  
→ successivamente corrispondono le signal → una con il mutex per rilasciare la risorsa e l'altra con piene per incrementare il numero di messaggi che sono pieni all'interno del buffer  
→ uno in più da essere consumato → se in particolare ci sarà un consumatore in attesa questo sarà svegliato e quindi potrà prelevare il dato

+in modo opposto il consumatore:

```

do {
 wait(piene); // decrementa n. posiz. piene e attende se <=0
 wait(mutex); // mutua esclusione con processi produttore

 //rimuove un elemento dal buffer

 signal(mutex);
 signal(vuote); // incrementa n. posizioni vuote,
 // eventualmente sveglia processo in attesa
 //consuma l'elemento rimosso
} while (true);

```

→ wait su piene per controllare che ci sia almeno un dato nel buffer → altrimenti se piene = 0 si aspetta → inizialmente se ad esempio faccio partire tutti i thread consumatori → questi saranno in attesa su wait(piene) → nel momento in cui viene inserito un valore dal produttore (attraverso la chiamata signal(piene)) → piene diventa 1 per un momento e solo uno dei tanti consumatori verrà svegliato per prendere il valore inserito → a quel punto lui ci può accedere → toglie l'elemento dalla coda e infine dà signal(vuote) per incrementare il numero di posizioni vuote

+Nel caso invece di memoria illimitata invece non servirà più il semaforo vuote perché non è possibile inizializzare ad N finito il semaforo... e inoltre non mi servirà più sapere il numero di posizione vuote

## Problema dei lettori e scrittori

Abbiamo un insieme di dati condiviso fra processi o thread concorrenti → dove abbiamo alcuni thread che leggono i dati, ma non effettuano modifiche  
→ mentre altri che scrivono e leggono i dati...

→ si vuole che più *lettori* possano leggere contemporaneamente il contenuto dei dati condivisi → siccome non li modificano non ci sono problemi se più thread leggono uno stesso dato → ma solo un thread alla volta possa leggere/scrivere bloccando tutti i lettori!  
→ quando un thread riceve l'accesso in scrittura tutti gli altri aspettano → sia i lettori che gli scrittori → anche loro non possono né leggere né scrivere

+Come realizzare questa sincronizzazione tramite semafori?

→ questi thread in particolare avranno:

- dati (condivisi)
- Un semaforo mutex → inizializzato a 1
- Un semaforo *scrittura* → anche lui inizializzato a 1 e indica se la scrittura può essere fatta (1) o altrimenti no (0) → si comporta in modo simile al mutex → infatti solo un processo alla volta può essere attivo sulla scrittura dati
- un intero → *nlettori* → inizializzato a zero che indica il numero di lettori che stanno leggendo in quel momento

+Riporto quindi i codici per i processi scrittori:

```

do {
 wait(scrittura); //aspetta nel caso scrittura sia 0

 // esegue lettura e scrittura

 signal(scrittura); //porta scrittura a 1, e risveglia
 //eventuale processo in attesa
} while (true)

```

→ nel caso di 2 scrittori che vogliono contemporaneamente scrivere solo 1 riesce ad entrare → quello che entra esegue il suo codice di lettura/scrittura e quando termina rilascia l'uso della risorsa → facendo entrare l'altro scrittore (o anche un altro lettore)

+ Per i lettori dice pierfra è un po' più complicato:

```

do {
 wait(mutex) ;
 nLettori ++ ; //incrementa numero processi lettori
 if (nLettori == 1) //se è il primo lettore
 wait(scrittura) ; //aspetta nel caso ci sia una scrittura in corso
 //altrimenti inibisce la scrittura
 signal(mutex)
 // esegue lettura
 wait(mutex) ;
 nLettori--; //decrementa numero processi lettori
 if (nLettori == 0) // se non ci sono più lettori attivi
 signal(scrittura); //sveglia un eventuale processo scrittore
 signal(mutex) ;
} while(true)

```

→ si usa un semaforo mutex per garantire che solo un lettore alla volta possa accedere alla variabile *nlettori* → in quanto comporta un incremento è se non sincronizzato in maniera adeguata può portare al problema della race conditions!

→ dopo quindi aver incrementato il numero di lettori → se è il primo lettore → blocca anche la scrittura(*inibisce* la scrittura → con la lock un altro processo scrittore non può accedere!)

→ oppure se c'è un processo scrittore che attualmente sta scrivendo aspetta!

→ finito questo fa la signal(mutex) sbloccando la risorsa → questo garantisce che solo un lettore alla volta può fare queste operazioni → il **primo** fra i lettori che riesce ad accedere blocca (se può) anche la scrittura! → lascia poi al secondo il quale non farà il blocco della scrittura ecc... tutti gli altri lettori infatti incrementano la variabile nlettori che dice quanti sono i lettori che stanno leggendo in quel momento!

→ prima di rilasciare la scrittura bloccano di nuovo il mutex → decrementano il numero di lettori e solo nel caso questo sia zero vuol dire che tutti i lettori hanno lasciato il loro blocco → allora in quel caso si può fare la signal su scrittura per svegliare un eventuale processo scrittore in attesa e infine fare signal mutex

+ “Francesco chiede: ma non voleva che i lettori potesse leggere in contemporanea??” → pierfra risponde: appunto! → il codice di eseguo lettura è eseguito in contemporanea! → si possono avere più letture attive contemporaneamente → viene serializzato l'accesso singolo → ma le lettura si possono fare in parallelo!

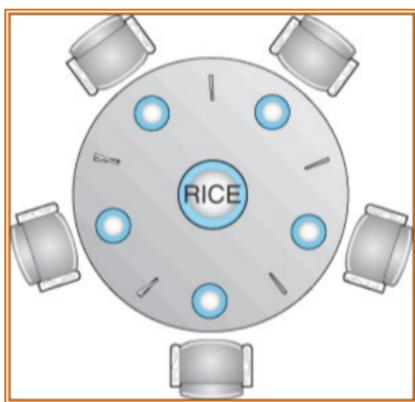
+ Consideriamo quindi il caso in cui c'è uno scrittore che sta scrivendo → questo essendo il primo e riuscito ad entrare tiene bloccato per tanto tempo la scrittura...

Ipotizziamo quindi che arrivino successivamente i lettori → il primo che riesce ad entrare porta il nLettore a 1 chiede la wait scrittura → ma scrittura è zero! → per cui il lettore si blocca! → avendo però chiesto il mutex e non riuscendo a chiamare la signal → anche tutti gli altri lettori che tentano di accedere alla struttura si trovano bloccati all'inizio con wait(mutex) ! → fintantoché lo scrittore tiene bloccato la risorsa tutti gli altri aspettano! → ed è proprio quello che si vuole! (uso della doppia wait!)

→ finita quindi la scrittura da parte dello scrittore → questo libererà chi è in attesa sulla scrittura → in particolare il primo thread lettore che aveva bloccato! → quindi anche lui fà le cose che deve fare (se c'è ne sono ma in genere no) e infine la signal(mutex) → liberando il secondo thread che era in attesa → e così via per tutti i thread che erano in attesa sulla wait(mutex) iniziano tutti a riuscire ad entrare nella fase iniziale di accesso alla struttura → per eseguire la lettura che può avvenire in parallelo!

→ finiti i lettori anche questi rilasciano uno alla volta la risorsa e così via

### Problema dei 5 filosofi (o dei 5 villabanks)



→ 5 persone con ognuno una *bowl* e una bacchetta → siccome il riso con una bacchetta sola non si riesce a prendere → si ha bisogno di almeno 2 bacchette!

→ ognuno dei filosofi quindi per mangiare ha bisogno sia della bacchetta a sinistra che di quella a destra! → devono quindi queste 2 risorse → ovviamente essendoci solo 5 bacchette qualcuno dovrà aspettare!

Come si implementa il tutto?

Abbiamo il piatto di riso centrale condiviso che è appunto un dato condiviso e poi abbiamo le 5 bacchette che funzionano da semaforo inizializzati a 1 → bacchetta che è usata o non è usata ma solo da una persona

Implementiamo quindi il codice per l'i-esimo filosofo:

```

do {
 wait(bacchetta[i]);
 wait(bacchetta[(i + 1) % 5]);
 //mangia
 signal (bacchetta[i]);
 signal (bacchetta[(i + 1) % 5]);
 // pensa
} while (true) ;

```

→ si acquisisce la bacchetta i-esima(bacchetta a sinistra) e poi quella i-esima+1 ovvero a destra (che è la successiva) → che nel caso dell'ultimo filosofo non avendo una bacchetta a destra, si ritornerà all'inizio! → ad esempio per 5 bacchette se l'i-esimo filosofo è il 4 → allora si prende la bacchetta 0! (*aritmetica modulare!*)

→ acquisite entrambe le bacchette → può mangiare → e una volta che ha fatto rilascia le bacchette → tramite le signal!

→ sembra tutto ok... ma in realtà c'è un problema! → cosa succede se tutti i filosofi vogliono mangiare insieme → dunque ognuno si prende la bacchetta sinistra che gli spetta → ma per la wait successiva ogni filosofo non riesce a prendere la bacchetta successiva! (se tutti riescono a prendere la prima bacchetta) → per cui si entra in uno stato di stallo! → tutti hanno acquisito la prima ma nessuno riesce ad acquisire la seconda! → i filosofi poverini non riescono a mangiare e muoiono di fame → situazione da evitare!

→ come risolvere il problema?? boh lo dirà dopo...

## Problemi con i semafori

Possono capitare dei problemi nell'uso dei semafori... se ad esempio uno inverte la signal con la wait → questo può portare ad avere più processi nella sezione critica → perchè al posto di decrementare... si incrementa prima

→ oppure fare una doppia wait sul mutex può portare allo stallo! → il secondo wait infatti rimane bloccato dal primo e nessuno lo sblocca! → wait(mutex)...wait(mutex) → autostallo  
→ oppure ancora uno si dimentica di chiamare la wait o la signal o entrambi → che può portare o allo stallo o a più processi nella sezione critica...

→ per cui l'uso delle wait e signal è potente, ma può essere pericoloso!

+Un altro uso non corretto è il seguente:

|                 |                   |
|-----------------|-------------------|
| wait(mutex)     | wait(mutex)       |
| ...             | ...               |
| wait(semaphore) | signal(semaphore) |
| ....            | ...               |
| signal(mutex)   | signal(mutex)     |

→ tra una wait e una signal su un semaforo... tra questi 2 richiedo un'altra wait su un altro semaforo → se questa wait è fatta su un semaforo che ha già valore 0 → allora si entra in attesa → ma l'unico che può svegliare il processo è una signal che però a sua volta risulta

bloccata in un altro processo dove è stata fatta una wait sul mutex! → per cui rimane tutto bloccato!

→ I problemi di sincronizzazione dovuti a errato uso dei semafori sono molto difficili da individuare in quanto si possono anche verificare solo in **rare** occasioni (dipende dai processi in esecuzione, dalla loro velocità) e difficilmente sono ripetibili!

→ per risolvere questo tipo di problemi (o evitarli) si usano i monitor → che sono un modo per rendere l'uso dei semafori più semplice in modo da evitare gli errori!

+Occhio a non scambiare le wait nei vari problemi di sincronizzazione! → ad esempio sui prod/consumatori → fare prima wait(mutex) e poi wait(vuote) → vuote si blocca solo quando il buffer è pieno e quindi solo nel caso in cui abbiamo produttori molto intensi che riempiono il buffer... negli altri casi non abbiamo l'effetto voluto!

+58 min in poi da reimplementare! (esempi in java dei problemi di sincronizzazione)  
Stampare significa accedere ad una risorsa condivisa... dunque rallenta le cose → cambiano di molto le performance (soprattutto nel caso prod-cons)

# 7/05/2021

Primi 24 min da reimplementare... (filosofi in java)

## I monitor

Meccanismo che permette di sincronizzare processi o thread → risolvono in qualche modo i problemi dovuti ai semafori → i quali sono molto potenti per cui c'è anche la possibile di fare grossi errori (anche non accorgendosene)

→ il monitor ha la caratteristica che solo **un processo alla volta** può essere attivo all'interno del monitor!

+Esempio:

```
monitor nome_monitor
{
 // dichiarazione variabili condivise
 procedure P1 (...){ }

 ...
 procedure Pn (...){.....}

 initialization code (....){ ... }

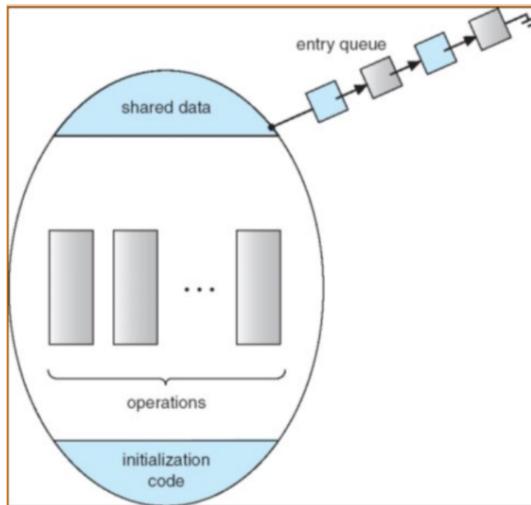
 ...
}
```

→ all'interno del monitor sono presenti determinate procedure che implementano le funzionalità del monitor → ci sarà una parte di dichiarazioni di variabili condivise fra le varie procedure e infine una parte di inizializzazione del codice!

→ la caratteristica è che solo un processo/thread alla volta può essere in esecuzione in una delle varie procedure!

→ in sostanza è come se fosse presente un mutex implicito che viene acquisito **prima** di poter entrare nel monitor → e sarà rilasciato nel momento in cui termina chi aveva acquisito l'accesso (poichè è un mutex fatto a livello di linguaggio vengono gestiti automaticamente tutti i casi... compreso quello di lancio delle eccezioni → il tutto risulta quindi più sicuro!)

+Una visione del monitor è la seguente:



→ abbiamo dei dati condivisi tra gli utilizzatori del monitor e poi abbiamo una coda dei processi che sono in attesa di entrare nel monitor!

→ se un processo si trova all'interno del monitor, sta eseguendo una procedura e contemporaneamente arriva un altro processo che richiede un servizio del monitor → per cui lui aspetta! (come se ci fosse un mutex a bloccare l'accesso alla risorsa)

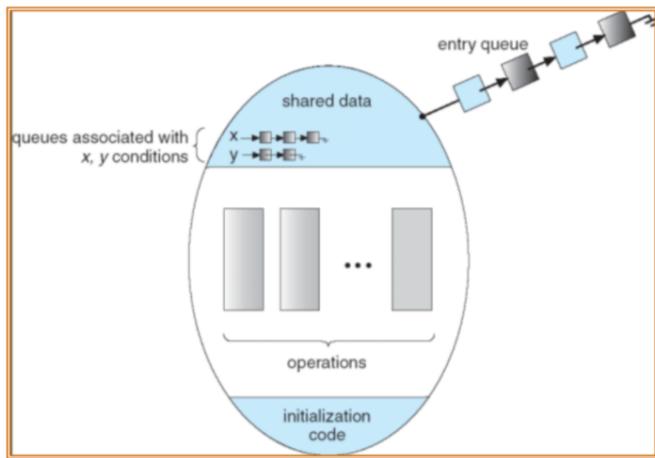
→ occhio che le procedure non sono dei processi ma dei semplici metodi!

+C'è però bisogno di qualcosa in più, oltre al fatto che i thread entrano uno alla volta nel monitor... questa richiesta viene soddisfatta dalle *variabili condizioni* → “condition x,y” → le quali possono essere definite all'interno di un monitor e hanno una particolare semantica → in particolare possono fare 2 cose:

- mettersi in attesa → siamo nel corpo di una procedura di un monitor e ad un certo punto, si decide che lui non può andare avanti ma si deve aspettare che accada qualcos'altro... aspetta che qualcuno gli segnali che sia accaduto qualcosa... (perfora molto vago) → lui quindi si mette in attesa su una certa condizione... chiamando il metodo “`x.wait()`” → il processo che invoca l'operazione viene sospeso e oltre a mettersi in attesa implicitamente lascia il monitor! → permettendo così ad altri processi di entrare e quindi di effettuare operazioni sul monitor! → fra queste operazioni ce ne sarà uno che sbloccherà il processo in attesa → e quindi lo sveglierà quando il dato è arrivato
- operazione “`x.signal`” → la quale serve appunto a svegliare il processo che si trova in attesa su quella condizione! → ovvero uno che processi che ha invocato `x.wait`!

+C'è un problema!! → quando viene eseguita la signal → si sveglia il processo in attesa di un certo evento per cui entrambi i processi vorranno eseguire l'istruzione successiva alla wait → e per fare questo dovrebbe rientrare nel monitor → il quale non è stato ancora liberato! → bisognerà quindi decidere chi dei 2 va avanti!! → il *segnalatore* o il *segnalato* → sarà quindi l'implementazione a decidere chi va avanti dei 2! → se viene data priorità a quello che era in attesa oppure viene data priorità a chi fà la signal!

+Schema di un monitor con le variabili condizionali:



→ notare che sono associate delle code → che saranno i thread che sono in attesa di quella condizione all'interno del monitor → al di fuori in coda invece ci sono i processi in attesa!

+Esempio: soluzione al problema dei 5 filosofi sfruttando i monitor!

```
monitor cinque_filosofi
{
 enum { PENSA, AFFAMATO, MANGIA } stato [5];
 condition self [5];

 void prendi(int i){
 stato[i] = AFFAMATO;
 verifica(i);
 if (stato[i] != MANGIA) self[i].wait();
 }

 void lascia(int i){
 stato[i] = PENSA;
 // controlla i filosofi vicini a destra e sinistra
 verifica((i + 4) % 5);
 verifica((i + 1) % 5);
 }
}
```

→ il monitor è uno e sarà condiviso fra tutti i processi che rappresentano i singoli filosofi  
 → la classe monitor ha quindi una variabile enum in cui viene memorizzato lo stato di ogni filosofo → 5 filosofi ognuno può trovarsi nello stato pensa - affamato - mangia(stato in cui il filosofo ha entrambe le bacchette e dunque riesce a mangiare)  
 +Si usa quindi un array di **condizioni** che rappresenta se stesso (?) → in pratica serve a bloccare/sbloccare se il filosofo riesce o meno ad acquisire entrambe le bacchette → quando non può come al solito... dovrà aspettare!  
 → a questo punto quando saranno disponibili le bacchette sarà segnalato e quindi si può procedere all'esecuzione

+Il monitor implementa 2 funzionalità che sono il *prendi* e il *lascia* → dove il parametro i è l'identificatore del filosofo  
 → dunque il metodo prendi per il filosofo i-esimo → imposterà lo stato a **affamato** a quel punto si richiama la funzionalità di *verifica*:

```

void verifica (int i){
 if ((stato[(i + 4) % 5] != MANGIA) &&
 (stato[i] == AFFAMATO) &&
 (stato[(i + 1) % 5] != MANGIA)){
 stato[i] = MANGIA;
 self[i].signal () ;
 }
}

```

```

initialization_code(){
 for (int i = 0; i < 5; i++)
 stato[i] = PENSA;
}

```

→ dove si controlla lo stato dei filosofi che stanno accanto all'i-esimo → se quello prima non sta mangiando (occhio all'aritmetica modulare) e lo stesso per quello dopo... vuol dire che entrambi o pensano oppure sono affamati... se quindi non hanno le bacchette → allora l'i-esimo può mangiare! → dunque pone lo stato a mangia e viene dato il signal!

→ si ritorna quindi al metodo prendi → se non si è riusciti a impostare lo stato a mangia allora si mette in attesa! (questo implica che dei suoi vicini almeno uno sta mangiando)

+si passa quindi al metodo lascia → dove sempre l'i-esimo filosofo, stavolta lascia le bacchette → si mette infatti nello stato pensa e successivamente fà la verifica di entrambi i suoi vicini! → nella verifica si ricontrolla che i vicini non stanno mangiando (i vicini del vicino preso come parametro) e se così il vicino può mangiare → verrà segnalato e quindi potrà procedere nell'esecuzione!

→ la verifica viene fatta su entrambi i vicini in modo che uno dei 2 possa andare avanti... oppure nessuno dei 2 nel caso i "vicini dei vicini" stiano a loro volta mangiando! (e quindi non possono accedere alle bacchette)

+Le condizioni sono pari appunto al numero dei filosofi e permettono di dire se un filosofo deve aspettare o meno! → si metterà in attesa sul proprio self (con il wait) nel caso lui non possa mangiare!

+L'implementazione in java dice pierfa sarà più semplice... cosa fondamentale però che si guarda i vicini (uno alla volta ?) e decide che può mangiare solo se riesce ad acquisire entrambe le bacchette! → non che ne guarda una e la acquisisce, ma che le guarda tutte e 2 e poi decide di prenderle entrambe oppure se ce n'è una sola sta lì ad aspettare! (lasciando la possibilità ad altri vicini di prenderla!) → questo ci garantisce il fatto di non avere stalli!

+Il programma principale sarà quindi:

```

monitor cinque_filosofi cf;

process filosofo(i)
{
 do{
 // pensa
 cf.prendi(i);
 // mangia
 cf.lascia(i);
 } while (true);
}
→ con tutte le fasi per ogni filosofo...

```

+I monitor sono presenti in vari linguaggi come java, C# e addirittura anche in pascal...  
→ in java si possono realizzare tramite i semafori! (semafori forniti dal S.O, mentre i monitor dal linguaggio!)

## Implementazione monitor tramite semafori

Si usa un mutex per garantire l'accesso al monitor da parte di un processo alla volta!  
→ ogni procedura del monitor verrà arricchita di una serie di istruzioni che permettono il controllo del monitor!

```

wait(mutex);
...
body of F
...
if (next_count > 0)
 signal(next);
else
 signal(mutex);

```

→ prima istruzione sarà quella di mettere un wait sul mutex → dunque quando viene chiamata una procedura F → si chiederà il wait → e successivamente si eseguono le istruzioni della procedura...

Come visto altre volte bisognerà rilasciare il monitor → si guarda quindi se c'è qualcun altro che deve entrare, altrimenti si farà il signal del mutex per dire di lasciare l'accesso ad altri!  
+il semaforo next ed il contatore next\_count (inizializzati a zero) servono a garantire l'accesso al monitor ad un solo processo nel caso di una signal su una condizione → è legata infatti all'implementazione della signal e della wait sulle condizioni

+Per ogni variabile condizione (a seconda di quante sono usate) sono presenti un semaforo x\_sem (associato alla condizione x) ed un contatore x\_count(che dice quanti sono in attesa su quella condizione!)  
→ Per cui eseguendo x.wait → (legge semplicemente i commenti)

|                     |                                                            |
|---------------------|------------------------------------------------------------|
| <b>Wait:</b>        | <b>x.wait()</b>                                            |
| x_count++;          | // incrementa n. di thread in attesa di x                  |
| if (next_count > 0) | // se c'è un thread che ha fatto signal, in attesa monitor |
| signal(next);       | // lo sveglia                                              |
| else                | // altrimenti                                              |
| signal(mutex);      | // sblocca il mutex                                        |
| wait(x_sem);        | // aspetta x                                               |
| x_count--;          | // decrementa n. thread in attesa di x                     |

→ notare come non ci sia problemi di race conditions in quanto sono all'interno di un monitor e qui si esegue un thread alla volta!!

→ si controlla quindi se ci sono o meno thread da svegliare all'interno del monitor → in quest'ultimo caso lascia il mutex in modo che qualcun'altro possa entrare (dove???)

→ dunque si fa una wait sul semaforo per mettersi in attesa → siccome x\_sem (?) e tale che il signal viene solo se x\_count è maggiore di zero... il processo entrerà in attesa!

→ Quando quindi qualche altra procedura farà la signal su questa stessa condizione... dal wait x\_sem verrà svegliato e quindi si farà x.count - per decrementare quanti sono in attesa su x...

+Per la signal invece...

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| <b>Signal:</b>     | <b>x.signal()</b>                                     |
| if (x_count > 0) { | // se c'è un thread in attesa di x                    |
| next_count++;      | // incrementa thread in attesa di entrare nel monitor |
| signal(x_sem);     | // segnala a un thread in attesa di x                 |
| wait(next);        | // aspetta di rientrare nel monitor                   |
| next_count--;      | // decrementa thread in attesa di entrare nel monitor |

→ la signal sulla condizione (fatta da un altro processo/thread) prima cosa verifica se c'è qualcuno in attesa → ovvero se count è positivo vuol dire che c'è qualcuno → allora viene incrementato il next count → a quel punto viene fatta la signal dell'x\_sem (ovvero del semaforo associato alla condizione) → in modo da svegliare quello che era in attesa... e a questo punto LVI si mette in attesa, in modo da evitare di avere 2 thread nel monitor in contemporanea!

→ dunque quando viene fatta una signal e questa sveglia un altro thread che era all'interno del monitor allora viene dato precedenza a quello che era in attesa! → proprio quello che ha fatto la signal si mette in attesa... lasciando l'uso del monitor ad altri → sempre LVI si mette in attesa per rientrare nel monitor(attesa con il next)

→ quando verrà svegliato → nel caso qualcun altro entri nel wait → ovvero l'unico che è nel monitor torna in wait → lui sarà svegliato!

+Oppure ancora se è presente uno solo(next\_count = 1) che era nel monitor e vuole uscire allora si farà la signal del next in modo da svegliare quello che era rimasto in attesa sulla signal!

→ come al solito nel momento in cui LVI verrà svegliato(metodo signal) si decrementa il count e poi si continua...

Cristianino (come me) non c'ha capito niente e chiede di rispiegarlo...

Il vincolo che abbiamo è che: un solo thread può essere all'interno di una delle procedure del monitor → solo uno può essere in esecuzione sul monitor

→ il resto che ci vuole accedere sono invece fuori, quando l'unico all'interno dunque entra in wait → perchè ad esempio vuole aspettare che venga prodotto un evento da qualcun'altro... per cui chiamando il metodo wait(), incrementa il contatore e si mette in attesa

→ nel caso però lui sia **solo** e nessun altro è in attesa (di continuare dopo una signal) → allora lui esce dal mutex e quindi si mette in attesa → lasciando l'uso del monitor a qualcun'altro !

→ arriva quindi un altro thread che esegue una procedura del monitor e questo porta alla signal dell'evento che il processo precedente stava aspettando → chiamata quindi la funzione di signal si vuole segnalare al processo precedente che può continuare → lo fa facendo la signal sul semaforo su cui sempre il processo precedente era in attesa → però il problema è che se il processo attuale continua l'esecuzione allora ci sono 2 processi all'interno del monitor! → quindi l'attuale si mette in attesa perché aveva svegliato l'altro! (forse l'altro è identificato con next)

→ il processo precedente dunque fà le sue cose e quando esce si ritorna al codice iniziale (prima immagine sotto implementazione monitor) → si controlla quindi se ci sono processi che sono in attesa (sì! → quello che gli aveva fatto la signal e stava aspettando di rientrare) → dunque si ritorna ad eseguire il processo attuale! → si decrementa il next\_count - - e si va avanti...

+Ma non finisce qua! → il processo precedente (si ancora lui) potrebbe aver fatto una wait su un'altra condizione → quindi si troverà con il next\_count > 0 (guarda metodo wait) → si risblocca quindi il processo "attuale" per continuare → il precedente si rimette in attesa mentre l'attuale procede!

+In realtà è tutto molto teorico... che è dunque associabile a vari linguaggi... ad esempio nel C che non sono presenti i monitor... dovremmo cercare di ricreare quel codice lì!

## Implementazione tramite Java

+Dopo tutta sta manfrina forse si passa alla vera implementazione in java dei monitor... Java implementa i monitor potendo definire i metodi **synchronized** → quando un metodo è dichiarato synchronized vuol dire che fà parte del monitor (ovvero è un metodo del monitor) Dunque nel caso di più thread che evocano l'uso di uno dei qualsiasi metodi sincronizzati → in realtà viene usato un mutex associato all'oggetto che impedisce di usare tutti i metodi sincronized di una particolare classe (eh???)

```
class SharedCount {
 private int count = 0;
 public synchronized void inc() {
 count++;
 }
 public synchronized int getCount() {
 return count;
 }
}
```

Se due o più thread usano lo stesso oggetto SharedCount ed entrambi chiamano il metodo inc() o getCount() solo uno alla volta possono incrementare count o accedere al valore di count

→ Ovvero se abbiamo un thread che è in esecuzione su un particolare metodo sincronizzato e c'è un altro thread che chiede l'accesso al metodo sincronizzato → quest'ultimo aspetterà! → solo un thread alla volta riesce ad entrare in uno qualsiasi dei metodi sincronizzati

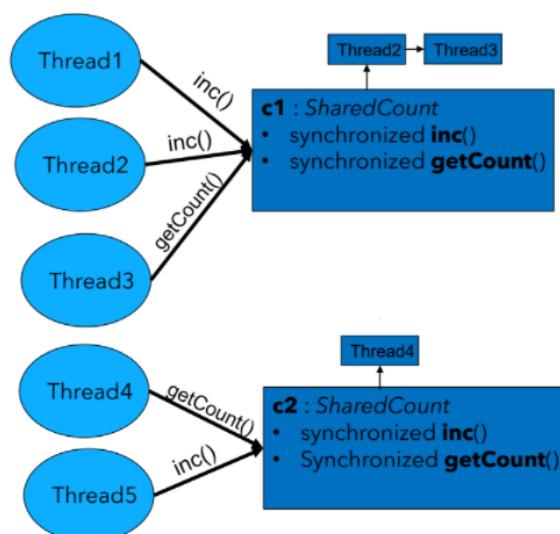
+Per cui chiamato un metodo sincronizzato verrà impostato il mutex e in qualsiasi caso si esca dal metodo, viene rilasciato il mutex! (anche in caso di eccezioni o return a metà)

+In realtà viene definito come lock *rientrante* → in particolare se da un metodo synchronized viene chiamato un altro metodo synchronized sullo stesso oggetto (altrimenti non vale) → allora non viene acquisito un nuovo lock → infatti il thread ce l'ha di già su quell'oggetto... quindi è possibile chiamare un altro metodo synchronized senza problemi!

→ il lock sarà rilasciato quando l'ultima esecuzione di un metodo sincronizzato (ovvero la prima chiamata → la quale blocca l'uso del monitor), viene fatta

+Anche i metodi statici possono essere definiti synchronized ma per loro non vi è associato nessun oggetto! → possono infatti essere chiamati anche senza un'istanza precisa di un oggetto! → in questo caso dunque si sincronizza l'accesso sulla base della classe → mutex associato alla classe e non alla singola istanza

+Esempio:



Abbiamo 3 thread che condividono l'istanza dello `sharedCount` → si ipotizza quindi che solo thread1 riesca ad eseguire il metodo `inc()`, mentre gli altri 2 rimangono fuori in attesa → in una lista dei thread associati che sono in attesa sullo stesso oggetto!

+Se ho poi altri 2 thread che condividono un'istanza di `sharedCount` e si nota che prima entra il thread5 mentre thread5 aspetta di entrare...

+Cosa succede se da un metodo sincronizzato chiamo un altro metodo sincronizzato??  
Se ad esempio il metodo `inc()` è implementato con il metodo `getCount()` → allora l'uso dell'`inc()` porta a bloccare l'oggetto ma non la chiamata fatta all'interno del metodo `inc()` → la quale non porta a bloccarlo nuovamente!

## Wait e Notify

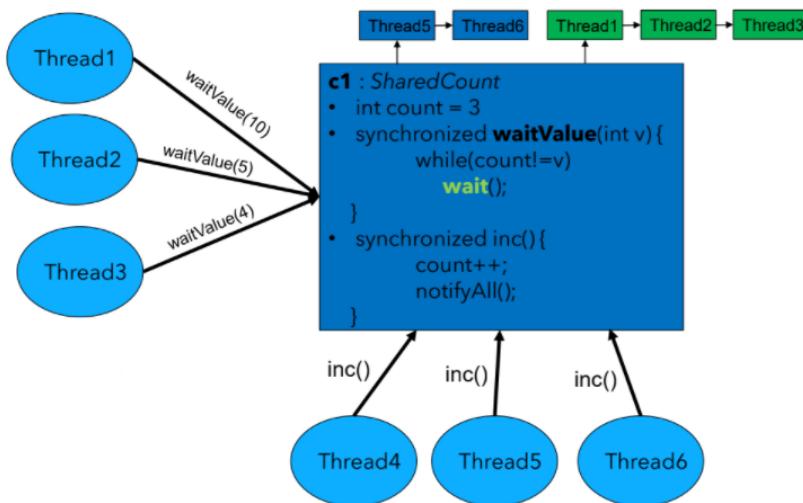
→ Come si gestiscono in Java le condizioni?? → ad un certo punto i metodi sincronizzati si dovranno mettere in attesa di qualcosa che qualche altro thread dovrà produrre... ovvero una condizione che qualche altro thread dovrà risolvere → in java si usano quindi i metodi wait e notify che sono appunto simili ai wait e signal

→ quando viene chiamato il metodo **wait** all'interno di un metodo sincronizzato → questo lascia il mutex (associato al metodo sincronizzato) e mette appunto il thread in attesa!  
 → aspetta quindi che sia chiamato un metodo del tipo notify o notifyAll che lo svegli

+Se un altro thread chiama il metodo **notify()** si estrae un thread in attesa sull'oggetto (spesso infatti sono informazioni legate al singolo oggetto e non alla singola condizione!)  
 → per cui si estrae un thread in wait (se presente) → questo viene sbloccato ma viene messo in attesa di poter acquisire il mutex! (sull'oggetto) → come al solito si avrebbe il problema di 2 oggetti attivi sul monitor... per cui si deve aspettare!  
 → quando quindi il metodo acquisirà l'accesso continuerà ad eseguire il codice dopo l'invocazione metodo **wait()**

+Il metodo **notifyAll()** è simile al **notify**, soltanto che alla sua invocazione vengono tolti dalla coda di attesa tutti i thread che erano in attesa sull'oggetto → per cui tutti tenteranno di entrare nel monitor e acquisire il lock dell'oggetto e poi piano piano tutti entreranno in esecuzione!

+Vediamo un esempio:



→ abbiamo 3 thread che chiedono di usare il metodo inc e i primi 3 thread invece che aspettano un valore (dunque una condizione) → ad esempio il primo aspetta che il valore del count arrivi a 10, il secondo 5 e così via...

→ il count viene incrementato ogni volta dal metodo **inc()** che viene invocato con i vari thread → dunque dopo aver incrementato il count si fa una chiamata al **notifyAll()** per dire che tutti quelli che sono in attesa sulla wait di sbloccarli e fai in modo che controllino lo stato → si risvegliano i thread 1-2-3 → uno di questi riuscirà ad entrare nel monitor quindi andrà a rivalutare la sua condizione! → infatti nel wait value ad esempio 4, si controlla lo stato del contatore → se è uguale a 4 bene... se non è uguale si mette in attesa!

→ per cui nel caso sia uguale a 4 riesce a terminare ed esce dal metodo sincronizzato → perchè è accaduto l'evento che stava aspettando!

→ notare quindi i thread in attesa di entrare nel monitor (in blu) e quelli in attesa di un qualche evento → che sono in attesa di una wait (in verde)

+Come si realizzano le attese sulle condizioni in java??

→ come visto nell'esempio si metterà un **while(! condizione)** → non condizione che stiamo aspettando e quindi ci mettiamo il **wait()** → occhio a non mettere l'**if** al posto del **while!!** → se

infatti ci fosse un if al risveglio dopo il notifyAll (ad esempio) i thread in attesa andranno ad eseguire l'istruzione successiva al wait() che in questo coincide con l'uscita dalla funzione!  
→ per cui tutti i thread si sbloccano come se avessero già raggiunto il valore che gli è stato assegnato! → sbagliato!!

+Mentre nel momento in cui la condizione richiesta risulta vera → si metterà in uno determinato stato attuale in modo tale da valutare se la condizione sia vera o falsa e a quel punto si può decidere se fare notify o notifyAll → quest'ultima è più sicura → infatti quello che era in attesa di quell'evento verrà svegliato! → e come visto nell'esempio abbiamo tre thread che sono in attesa di condizioni diverse! → per cui il notify avrebbe preso uno dei tre thread (ad esempio il thread 1) e lo avrebbe svegliato! ma solo lui! → per cui anche se ad esempio il count è già arrivato a 4 ma con il thread uno si ha la condizione di value = 10, allora non succede nulla e si rimane ancora in attesa!

→ nel caso invece tutti aspettino la stessa cosa e allora chiunque venga svegliato va bene!  
→ per cui va bene anche il notify()

+Bisogna stare attenti a non abusare dell'uso del notifyAll perché risveglio tutti i thread che sono in attesa! → e se avessi mille thread tutti in attesa gli risveglio tutti e ciascuno proverà ad entrare nel monitor! → tutti che usano la cpu per fare le operazioni e controllare la propria condizione e quindi l'uso della cpu non è proprio ottimo!

### Istruzioni synchronized

+C'è anche la possibilità di avere solo delle istruzioni sincronizzate → sincronizzare solo alcune istruzioni e non tutto un metodo! → infatti potrebbe esserci metodi molto lunghi e pesanti nei quali solo una piccola parte deve essere eseguita in modo che solo un thread alla volta possa eseguire quella parte di codice → ad esempio un race condition su un count++ può essere conveniente sincronizzare solo quella parte lì e non il resto!

→ la sintassi è:

```
synchronized(object) {
 istruzioni...
}
```

→ come parametro l'oggetto su cui si vuole sincronizzare → le istruzioni in questa parte di codice saranno eseguite solo se è stato possibile accedere al mutex associato a quel oggetto → il mutex sarà rilasciato appena le istruzioni saranno terminate → per cui si eseguiranno le altre istruzioni (se presenti) normalmente

→ all'interno del blocco potremmo chiamare i metodi di wait, notify e notifyAll, operando sull'oggetto specifico! → un metodo sincronizzato *m* è equivalente a qualcosa del tipo:

```
void m() {
 synchronized(this) {
 ...istruzioni del metodo...
 }
}
```

→ anche nella signature del metodo ci sarà il fatto di essere synchronized...

+Esempio di monitor in java(da riguardare... 1:30:00)

- quale scegliere fra semafori e monitor?? boh → pierfra dice che questi ultimi siano più veloci... anche se nel caso di un sacco di thread e facciamo la notifyAll diventa molto dispendioso per la CPU...
  - anche il fatto di avere del codice scritto con i semafori risulta poi difficile tradurlo in codice con i monitor... (notare che inizialmente java non aveva i semafori ma solo i metodi sinc)
- +se vuoi usare wait e notify → ricordarti di mettere nella signal del metodo synchronized!!

## 12/05/2021

Lezione di solo esercizi(da riguardare da 20min se vuoi...)

## 14/05/2021

### Gestione dello stallo

Siamo nella situazione in cui un insieme di processi risultano bloccati in quanto ognuno è in possesso di una risorsa e attende di acquisire una risorsa la quale è però posseduta da un altro processo, il quale a sua volta è in attesa di un ancora un'altra risorsa posseduta da un altro processo... e così via fino a formare un ciclo!

→ ogni processo dell'insieme risulta essere in attesa di un qualche altro processo posseduto da qualcun'altro dei processi in stallo

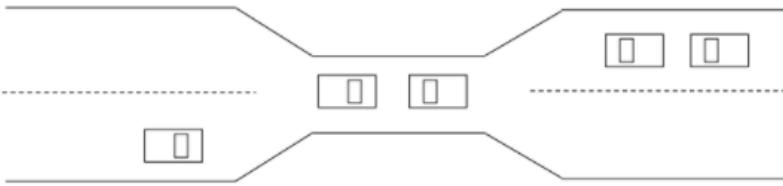
+Esempio: un sistema composto da 2 unità nastro → abbiamo P1 e P2 che hanno un unità ma hanno bisogno l'una dell'altra unità → ognuna quindi aspetta l'altro e si rimane in stallo

+Un altro esempio è quello dei semafori → sono presenti 2 semafori mutex inizializzati a 1  
→ solo un processo alla volta può utilizzare il mutex

|          |         |
|----------|---------|
| $P_0$    | $P_1$   |
| wait(A); | wait(B) |
| wait(B); | wait(A) |

→ se abbiamo 2 processi dove uno fà wait A e dopo wait B, mentre l'altro fà wait B e dopo wait A i due processi possono entrare in stallo! → la situazione è infatti analoga al nastro dove uno aspetta l'altro continuando all'infinito ad aspettare...

+Esempio del ponte(non gabry):



→ solo una macchina alla volta può accedere alla risorsa, che risulta essere il ponte e solo una delle 2 direzioni può accedervi!

→ se accade uno stallo, la situazione può essere risolta solo se una delle 2 macchine torna indietro! (altrimenti si aspetta → attesa indefinita)

### Modello del sistema

Proviamo quindi ad astrarre il problema costruendo un modello del sistema:

Abbiamo dei tipi di risorse  $R_1, R_2, \dots, R_m \rightarrow m$  diversi di risorse

→ di ogni risorsa  $R_i$  abbiamo  $W_i$  istanze → ma per ogni istanza abbiamo solo un uso esclusivo

→ infine ognuno usa un determinata risorsa seguendo certi passi:

→ Richiesta → Uso → Rilascio

+Diamo quindi una certa caratterizzazione per quando può avvenire uno stallo:

Lo stallo può avvenire se si hanno 4 condizioni contemporaneamente

(condizioni necessarie per avere situazione di stallo):

- **Mutua esclusione:** solo un processo alla volta può usare una risorsa.
- **Possesso e attesa:** un processo in possesso di una o più risorse è in attesa di acquisire altre risorse possedute da altri processi. → deve quindi possedere e volere accedere ad un'altra risorsa
- **No prelazione:** una risorsa può essere rilasciata solo volontariamente dal processo che la possiede quando ha finito di usarla → solo tramite rilascio del processo
- **Attesa circolare:** esiste una sequenza  $\{P_0, P_1, \dots, P_n\}$  di processi in attesa tale che  $P_0$  è in attesa di una risorsa posseduta da  $P_1$ ,  $P_1$  è in attesa risorsa posseduta da  $P_2, \dots, P_{n-1}$  è in attesa di una risorsa posseduta da  $P_n$ , e anche  $P_n$  è in attesa di una risorsa posseduta da  $P_0$ !

→ se c'è stallo sicuramente si hanno queste condizioni! → se ad esempio non si ha attesa circolare non si avrà stallo!

### Grafo di allocazioni delle risorse

Preso un insieme di vertici  $V$ , e un insieme di archi  $E$

→ i vertici sono quindi partizionati in 2 insiemi:

- $P = \{P_1, P_2, \dots, P_n\} \rightarrow$  insieme di tutti i processi nel sistema
- $R = \{R_1, R_2, \dots, R_m\} \rightarrow$  insieme di tutti i tipi di risorsa del sistema

→ allo stesso modo abbiamo 2 tipi di archi:

- **Arco di richiesta** → archi che vanno dal processo  $P_i \rightarrow R_j$  alla risorsa → e indica che il processo  $i$ -esimo sta richiedendo la risorsa  $j$ -esima → "è in attesa di..."
- **Arco d assegnazione** → archi opposti che vanno dalla risorsa  $R_j \rightarrow P_i$  al processo → e indica che un'istanza della risorsa  $j$ -esima è stata assegnata al processo  $P_i$

→ non esistono archi che vanno tra processi e archi che vanno tra risorse → ci sono solo archi da processi a risorse e viceversa

+Rappresentazione grafica:

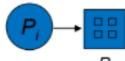
- Processo



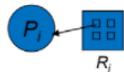
- Tipo di risorsa con 4 istanze



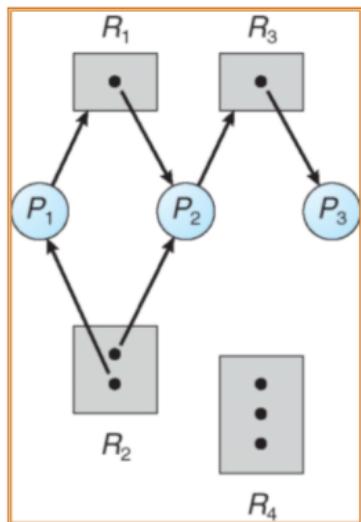
- $P_i$  richiede una istanza di  $R_j$



- $P_i$  possiede una istanza di  $R_j$

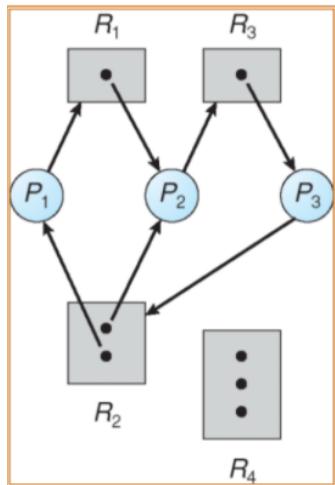


+Esempio: (situazione di non stallo)



→ abbiamo 3 processi e 4 tipi di risorse, di cui R1 e R3 hanno una sola istanza, mentre R2 e R4 tre istanze → P1 chiede un'istanza di R1, ma possiede un'istanza di R2 e così via gli altri... poiché anche il processo P3 non risulta in attesa di un risorsa chiesta da un altro processo non siamo in stallo!

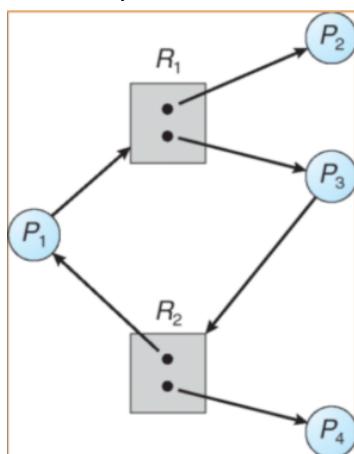
→ cosa che accade invece in questo caso:



--> P3 infatti richiede un'istanza di R2 che è già stata assegnata ad altri processi → per cui deve aspettare!

→ diversamente nella situazione precedente bastava che P3 rilasciasse l'istanza di R3, che dunque diventava disponibile per P2 → che quindi rilascerà successivamente le risorse a lui assegnate e in particolare l'istanza di R1 che sarà poi disponibile per P1! → non situazione di stallo!

+Occhio però che non sempre per una situazione di ciclo siamo in stallo!



→ in questa situazione necessariamente P1 e P3 aspetteranno che si liberino le risorse di R1 o R2 → ma in questo caso basta che o P2 o P4 rilascino le risorse per sbloccare la situazione! → non siamo quindi in stallo in quanto la situazione d'attesa può essere sbloccato da un evento prodotto da altri processi!

→ nel caso invece sia R1 che R2 abbiano solo un'istanza della risorsa allora ci sarebbe stato qualche problema...

+Alcuni fatti:

→ nel caso il grafo non abbia alcun ciclo, sicuramente non saremo in una situazione di stallo  
→ nel caso invece il grafo abbia un ciclo allora si hanno 2 situazioni:

- se è presente una sola istanza per ogni tipo di risorsa allora c'è stallo → diventa quindi una condizione **sufficiente** per la presenza di stallo
- se invece si ha più di una istanza per ogni tipo di risorsa allora c'è **possibilità** di stallo!

## Metodi per la gestione dello stallo

Si hanno diverse possibilità...

- **Prevenire o evitare** situazioni di stallo in modo che il sistema non entri mai in stallo.
- **Permettere** che il sistema entri in stallo, individuarlo e poi tenta di uscire da questa situazione → ad esempio facendo terminare un processo, chiamare un rollback ecc.
- **Ignorare** il problema, perchè ad esempio non gli riguarda → lasciando a livello dello sviluppo dell'applicazione per poi gestire eventuali situazioni di stallo  
→ Questa è la 'soluzione' di molti sistemi operativi tra cui Linux e Windows!  
→ problema che non viene risolto a livello di sistema operativo ma a livello di applicazione

## Prevenzione dello stallo

Abbiamo visto prima le condizioni necessarie allo stallo → per cui basterà rendere sicuramente falsa una delle condizioni e allora **non avremo mai** una situazione di stallo!  
→ alcune condizioni saranno più difficili rispetto ad altre di renderle false...

Ad esempio escludere la **mutua esclusione** non è semplice → ci sarà infatti un motivo se le risorse vengono usate in mutua esclusione → ad esempio nel caso concorrente ci saranno problemi → se quindi si riesce a condividere le risorse, la mutua esclusione non diventa più necessaria...

+Un'altra possibilità sarà quella di impedire **possesso e attesa** → ovvero garantire che quando si richiede una risorsa il processo non possieda altre risorse!  
→ se infatti un processo nel momento in cui chiede una risorsa, non ne ha nessuna, allora sicuramente non saremo in una situazione di stallo! → in quanto nessun altro può essere in attesa di qualcosa che ha in possesso lui  
→ si possono quindi definire dei *protocolli* di richiesta delle risorse → in modo che o chieda tutto all'inizio (rilasciandole tutte alla fine) oppure richieda le risorse solo quando **non le possiede!** → il problema sarà però nel caso in cui un processo prende tutte le risorse e le lascia solo dopo molto tempo → le usa per più tempo rispetto a quello che sarebbe necessario! → riducendo la possibilità ad altri di usare le risorse → *possibile attesa indefinita!*  
→ poichè inoltre tutti i processi vogliono prendere tutte le risorse insieme, e se non sono tutte disponibili allo aspetteranno... allora queste risorse non vengono efficacemente usate... ma ci saranno molti tempi di attesa in cui non sono sfruttate! (cosa che non avviene se i processi usano le risorse solo per il loro tempo necessario! → risorse usate in modo più efficiente)

+Un'altra possibilità è quella di **non usare la prelazione** → se un processo in possesso di alcune risorse richiede una risorsa non disponibile allora tutte le risorse possedute vengono rilasciate! → gli vengono tolte tutte in modo che le possano usare gli altri!  
→ le risorse prelazionate sono aggiunte alla lista delle risorse per cui il processo è in attesa  
→ il processo ripartirà solo quando potrà acquisire tutte le sue vecchie risorse più le nuove che sta richiedendo.

+Un ultima possibilità è quella di impedire un **attesa circolare** → imporre un ordinamento totale a tutti i tipi di risorse e imporre che ogni processo richieda le risorse in ordine crescente → siccome ho un ordinamento totale → ogni risorsa è presa in un certo ordine e quindi non sarà possibile avere un ciclo! (seguendo l'ordine di crescenza delle risorse non posso tornare indietro...)

## Evitare lo stallo

In questo caso risulta necessario che il sistema possieda delle informazioni a priori sull'uso delle risorse da parte dei processi

- il modello più semplice prevede che ogni processo dichiari il numero massimo di risorse di ogni tipo di cui potrebbe avere bisogno
- un particolare algoritmo dunque per evitare lo stallo esamina dinamicamente lo stato di allocazione delle risorse per assicurare che non ci potrà mai essere una condizione di **attesa circolare** → dove lo stato di allocazione delle risorse è definito dal numero di risorse disponibili e allocate e dalle richieste massime dei processi.

+Quello che si vuole fare è quello di garantire che il sistema si trovi in uno **stato sicuro** → ovvero sicuramente privo di stallo!

→ Quando il processo chiede una risorsa deve garantire che da uno stato sicuro, con la nuova richiesta si rimane in uno stato sicuro

→ un sistema è quindi detto sicuro se esiste un sequenza di stati sicuri di tutti i processi

La **sequenza**  $\langle P_1, P_2, \dots, P_n \rangle$  è **sicura** se per ogni  $P_i$ , le risorse che  $P_i$  può ancora richiedere possono essere soddisfatte con le risorse attualmente disponibili + le risorse possedute da tutti i processi  $P_j$ , con  $j < i$ .

→ se  $P_1$  può risolvere le richieste future usando solo le risorse disponibili → *bene*

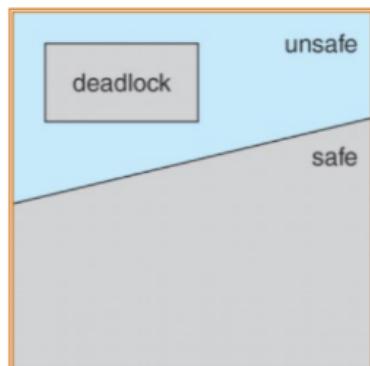
→ allo stesso modo  $P_2$  dovrà risolvere le richieste usando sia le risorse disponibili più quelle che ha  $P_1$ ! → quest'ultimo sicuramente lo riusciamo a sbloccare e  $P_2$  riesce a terminare usando le risorse liberate da  $P_1$  + quelle prima disponibili → e così via fino a  $P_n$  → dove tutti i processi riescono a terminare dunque non siamo in una situazione di stallo!

→ se ad esempio le necessità di  $P_i$  non sono disponibili, allora questo può aspettare fino a che tutti i  $P_j$  abbiano finito (con  $P_j$  processi precedenti all' $i$ -esimo)

→ Quando  $P_i$  termina,  $P_{i+1}$  può ottenere le risorse necessarie e così via...

+Ripetiamo quindi che se esiste una sequenza sicura allora potremmo dire di non essere in stallo, il problema si sposta quindi a trovare una sequenza sicura → ce ne possono essere diverse... ma basta trovarne una!

+Il vincolo di essere in uno stato sicuro è però più forte di quello di non avere stallo → infatti se un sistema è in uno stato sicuro sicuramente non avrà stallo, mentre nel caso di sistema non sicuro si ha la possibilità di uno stallo (e non la sicurezza!)



- dove con deadlock si indica la situazione di stallo! (stato non-sicuro != deadlock ma deadlock è uno stato non-sicuro)
  - se dunque riesco a impedire quelle transizioni che mi portano ad uno stato non sicuro, rimanendo sicuro → allora sicuramente eviterò lo stallo!
- +Come sottolineato prima risulta più difficile prevenire rispetto ad evitare la situazione di stallo → in quanto spesso si andrebbe a limitare le funzionalità del sistema! → o comunque ponendo dei vincoli che sono difficili da realizzare e garantire...

### Algoritmo con grafo di allocazione delle risorse

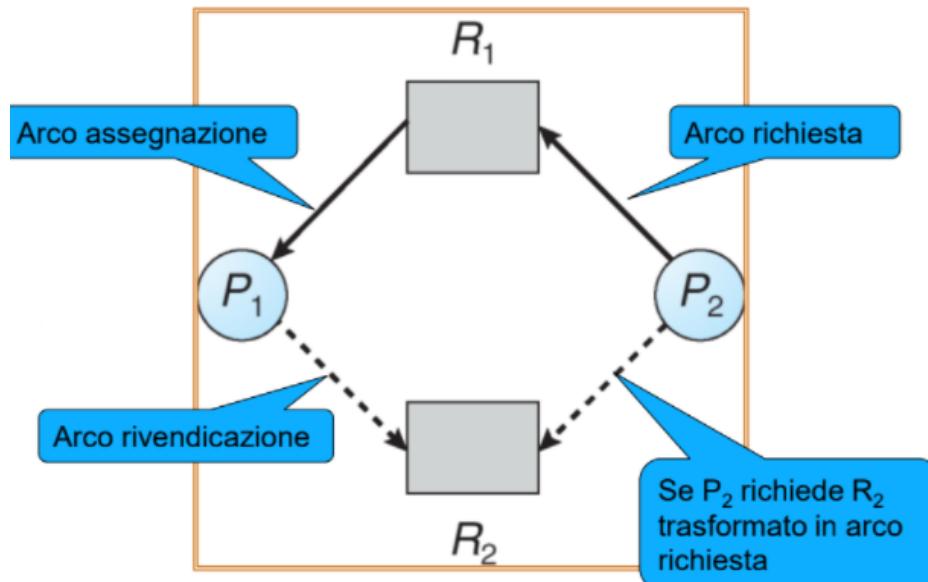
Si aggiunge un arco in più detto *arco di rivendicazione* → indica che il processo  $P_i$  potrebbe richiedere la risorsa  $R_j$  → e si rappresenta con un arco tratteggiato!

- l'arco di rivendicazione può essere convertito in un arco di richiesta quando quel processo li esegue effettivamente la richiesta
- quando invece la risorsa viene rilasciata dal processo allora l'arco di assegnazione passa a arco di rivendicazione → infatti tutte le risorse devono essere rivendicate a priori dal sistema → fatto stabilito all'inizio durante la dichiarazione del processo!

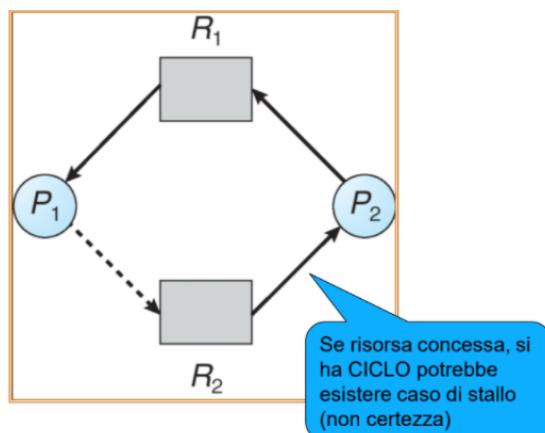
→ l'algoritmo prevede quindi che una risorsa venga concessa se dopo la sostituzione di un arco da rivendicazione ad assegnazione non si forma un ciclo nel grafo!

- se invece si forma un ciclo la risorsa non viene concessa in quanto potrebbe far entrare in una situazione di stallo!

+Il costo dell'algoritmo che opera su di un grafo è  $O(n^2)$



- Si presume che  $P_1$  e  $P_2$  possano richiedere in futuro una risorsa di  $R_2$ ! (all'inizio il grafo ha solo archi di rivendicazione → passano quindi ad archi di richiesta o assegnazione a seconda di cosa accade!)
- finchè un arco non diventa di assegnazione la risorsa è ancora libera di essere usata → dunque l'arco può diventare di assegnazione!



→ come scritto se assegnata la risorsa a P2 si potrebbe formare un ciclo → se infatti P1 richiede R2 in futuro questo porta ad avere stallo! (potenzialmente si potrebbe entrare in stallo)

→ dunque per evitare questa situazione non si concede la risorsa di R2 a P2! (questo nel caso R2 abbia una singola istanza per la risorsa)

### Algoritmo del banchiere

Questo è invece il caso di **più istanze** di risorse per tipo → quindi si usa un particolare algoritmo detto del *banchiere*

→ Ogni processo deve dichiarare a priori il massimo uso per ogni tipo di risorsa e quando un processo richiede una risorsa potrebbe dover attendere!

→ infine quando un processo prende tutte le risorse necessarie le deve ritornare comunque in tempo finito

+Quali strutture dati usa l'algoritmo?

→ Presi n processi e m tipi di risorse, abbiamo un vettore detto degli **available** → lungo m e per ogni risorsa viene detto quante di quelle risorse sono in un certo momento disponibili!

→ Se Available[ j ] = k, ci sono k istanze della risorsa Rj disponibili (con  $k \geq 0$ )

→ poi è presente una matrice **Max** di dimensioni  $n \times m$ , dove alla posizione Max[i,j] = k, e vuol dire che il processo i-esimo potrebbe richiedere al massimo k istanze della risorsa Rj! → matrice dunque costruita inizialmente sulla base dei processi che sono in esecuzione e quante risorse al massimo vogliono usare

→ poi è presente un'altra matrice delle allocazioni → **Allocation**  $n \times m$  → che rappresenta lo stato di allocazione dei processi verso le risorse → per cui se Allocation[ i ,j ] = k allora Pi ha attualmente allocate k istanze di Rj → questa matrice serve quindi a tenere lo stato del sistema

→ infine è presente la matrice **Need** → che rappresenta il numero delle risorse che ogni processo potrà utilizzare in futuro → di quante ne ha bisogno in futuro per terminare! → Se Need[ i,j ] = k, allora Pi potrebbe richiedere altre k istanze di Rj per completare il task.

In particolare si calcola come  $Need [i,j] = Max[i,j] - Allocation [i,j]$ .

### Algoritmo di verifica

1. Presi quindi 2 array Work e Finish, di lunghezza rispettivamente m e n → work tiene l'occupazione delle risorse e finish indica se quel processo lì è terminabile → ovvero può terminare con le risorse presenti → i 2 vettori sono quindi inizializzati come Work = Available → con work che tiene quante risorse ci sono a disposizione nel

sistema in un certo momento, mentre finish è inizializzato tutto a false →  $\text{finish}[i] = \text{false}$  per  $i = 1, \dots, n$  (si presume che nessuno dei processi possa essere terminato!)

2. Trova un processo  $i$  tale che:

→  $\text{Finish}[i] = \text{false} \rightarrow$  ancora non siamo riusciti a stabilire se lui termina o meno  
→ e per quel processo  $\text{Need}(i) \leq \text{Work} \rightarrow$  numero di risorse richieste è minore o uguale al numero di risorse disponibili in quel momento → in questo caso dunque il processo  $i$ -esimo riesce a terminare! (se  $\text{finish}[i] = \text{true}$  allora risulterebbe inutile guardare se riesce a terminare!)  
→ si guarda dunque sulla base degli available se c'è qualcuno che può terminare!  
→ se quindi non lo troviamo saltiamo direttamente al passo 4... se invece lo trovo vado al passo successivo (passo 3)

3. Al work aggiungiamo le risorse che lui possiede →  $\text{work} = \text{work} + \text{Allocation}(i) \rightarrow$  risorse che possono quindi essere usate da altri processi per terminare! → inoltre viene marcato che quel processo lì può terminare come  $\text{finish}[i] = \text{true} \rightarrow$  a quel punto si ritorna al passo 2 → si va alla ricerca di un altro processo non ancora impostato a true che però abbia  $\text{need}(i) \leq \text{work}$  → risorse richieste inferiore alle risorse che al momento sono disponibili! → il tutto serve quindi per capire se siamo in uno stato sicuro o meno!! (il tutto ovviamente è simulato... le risorse non vengono ne prese ne rilasciata effettivamente) + $\text{need}(i)$  è la riga  $i$ -esima della matrice

4. Arrivo quindi o già dal passo 2, oppure se per ogni processo  $i$  ho trovato che le risorse richieste sono minori di quelle disponibili → allora vado dal passo 3 e controllando se  $\text{Finish}[i] == \text{true}$  per tutti gli  $i \rightarrow$  allora in questo caso il sistema si trova in uno **stato sicuro!!** → altrimenti se non è vero anche per un solo processo **non** sono in uno stato sicuro! (notare che nel caso i processi siano tutti veri per il  $\text{finish}$  allora non ripasserò dal passo 3 e andrò subito al 4 per terminare l'algoritmo!)

+Notare che  $\text{Need}(i)$  e  $\text{Allocation}(i)$  indicano la riga  $i$ -esima della matrice → e quindi è il vettore di allocazione per il processo  $i$ -esimo!

→ l'algoritmo dunque serve a verificare se il sistema si trovi in uno stato sicuro → facendo la verifica dello stato! → con questo algoritmo dunque saremo in grado di decidere se dare o meno la risorse!

Algoritmo di richiesta risorse per il processo  $P_i$

Passato il test di stato sicuro, un processo fa la richiesta → abbiamo quindi un vettore di richieste → detto **Request[i]** → e il processo  $i$ -esimo chiede un certo numero di istanze della risorsa  $R_j \rightarrow$  parte così l'algoritmo:

1. Se il vettore  $\text{Request}(i) \leq \text{Need}(i)$  vai a passo 2 → *richiesta valida* → ovvero ok e prosegui nell'algoritmo. Altrimenti, c'è un errore dal momento che il processo richiede più risorse di quelle dichiarate! → probabilmente il processo ha sbagliato a dare il massimo! → in quel caso si blocca e si riporta un errore del sistema (ad esempio?)
2. Se  $\text{Request}(i) \leq \text{Available}$ , vai a passo 3 → questo sta ad indicare che le risorse richieste sono effettivamente disponibili... Altrimenti  $P_i$  deve aspettare, perché le risorse non sono (ancora) disponibili!
3. Arrivati a questo punto le risorse richieste sono giuste e sono disponibili, dunque si *simula* la concessione delle risorse e si verifica se lo stato sia sicuro o meno! → se concedendo le risorse al processo, il nuovo stato in cui ci troviamo è uno stato sicuro o meno! → la simulazione viene quindi fatta come:  
 $\text{Available} = \text{Available} - \text{Request}(i); \rightarrow$  diminuisco le risorse disponibili  
 $\text{Allocation}(i) = \text{Allocation}(i) + \text{Request}(i); \rightarrow$  aumento l'allocazione per il processo  $i$

$\text{Need}(i) = \text{Need}(i) - \text{Request}(i)$ ; e riduco le need sempre del processo i-esimo!

→ faccio una simulazione attraverso una copia, dove aggiorno le matrici → a questo punto applico l'algoritmo di verifica (visto prima) per capire se il nuovo stato sia sicuro o meno! → se concedendo le risorse il sistema si trova in uno stato per cui tutti i processi riescono a terminare!

→ Se dopo il cambiamento ho uno **stato sicuro** → le risorse sono allocate a Pi.

→ Viceversa se **stato unsafe** → allora Pi deve aspettare! → il vecchio stato allocazione risorse deve essere rispristinato e dunque in sostanza non si modifica lo stato!

### +Esempio

- 5 processi  $P_0 - P_4$ ; 3 tipi risorsa:  
 $A$  (10 istanze),  $B$  (5 istanze), and  $C$  (7 istanze).
- Snapshot al tempo  $T_0$ :

|       | <u>Allocation</u> |          |          | <u>Max</u> |          |          | <u>Available</u> |          |          |
|-------|-------------------|----------|----------|------------|----------|----------|------------------|----------|----------|
|       | <u>A</u>          | <u>B</u> | <u>C</u> | <u>A</u>   | <u>B</u> | <u>C</u> | <u>A</u>         | <u>B</u> | <u>C</u> |
| $P_0$ | 0                 | 1        | 0        | 7          | 5        | 3        | 3                | 3        | 2        |
| $P_1$ | 2                 | 0        | 0        |            | 3        | 2        |                  |          |          |
| $P_2$ | 3                 | 0        | 2        |            | 9        | 0        | 2                |          |          |
| $P_3$ | 2                 | 1        | 1        |            | 2        | 2        |                  |          |          |
| $P_4$ | 0                 | 0        | 2        |            | 4        | 3        | 3                |          |          |

→ la situazione iniziale dunque risulta per  $P_0$  che ha una sola risorsa di tipo B, ma potrebbe chiedere al massimo 7 risorse di tipo A, 5 di tipo B e 3 di tipo C e così via il resto dei processi... infine ci sono disponibili (al momento) tre risorse di tipo A e di tipo B, mentre 2 di tipo C! → lo stato descritto da queste matrici... sarà sicuro??

→ si calcola la matrice del Need come  $\text{Need}(i) = \text{Max}(i) - \text{Allocation}(i)$  → differenza riga per riga!

|       | <u>Allocation</u> |          |          | <u>Max</u> |          |          | <u>Need</u> |          |          | <u>Available</u> |          |          |
|-------|-------------------|----------|----------|------------|----------|----------|-------------|----------|----------|------------------|----------|----------|
|       | <u>A</u>          | <u>B</u> | <u>C</u> | <u>A</u>   | <u>B</u> | <u>C</u> | <u>A</u>    | <u>B</u> | <u>C</u> | <u>A</u>         | <u>B</u> | <u>C</u> |
| $P_0$ | 0                 | 1        | 0        | 7          | 5        | 3        | 7           | 4        | 3        | 3                | 3        | 2        |
| $P_1$ | 2                 | 0        | 0        |            | 3        | 2        | 1           | 2        | 2        |                  |          |          |
| $P_2$ | 3                 | 0        | 2        |            | 9        | 0        | 6           | 0        | 0        |                  |          |          |
| $P_3$ | 2                 | 1        | 1        |            | 2        | 2        | 0           | 1        | 1        |                  |          |          |
| $P_4$ | 0                 | 0        | 2        |            | 4        | 3        | 4           | 3        | 1        |                  |          |          |

→ si controlla quindi se con le risorse disponibili qualche processo può essere terminato! → ovvero l'available soddisfa il need di qualcuno? → Si! →  $P_1$  può essere soddisfatto! → dunque potrà rilasciare le risorse di tipo A che possiede → avremo quindi una nuova situazione di work con le risorse disponibili  $[5, 3, 2]$  e si riparte da capo con l'algoritmo!

- $\text{Work} = [3 \ 3 \ 2] \rightarrow +A_1 \rightarrow [5 \ 3 \ 2] \rightarrow +A_3 \rightarrow [7 \ 4 \ 3] \rightarrow +A_4 \rightarrow [7 \ 4 \ 5] \rightarrow +A_2 \rightarrow [10 \ 4 \ 7] \rightarrow +A_0 \rightarrow [10 \ 5 \ 7]$

→ per cui la sequenza dei processi  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  costituisce una sequenza sicura → si nota ovviamente che in alcuni casi potevo terminare direttamente già altri processi per cui quella trovata è solo una delle varie sequenze sicure presenti! → basta comunque che ne trovi una...

+Simuliamo dunque la richiesta:

- supponiamo ad esempio che il processo P1 richieda (1,0,2), cosa può succedere?
- se gli diamo la risorsa saremo in grado di rimanere comunque in uno stato sicuro?

Controllo quindi inizialmente se la request risulta minore dell'available → e in questo caso  $(1,0,2) \leq (3,3,2)$  → quindi potenzialmente è disponibile!

→ si aggiorna il need e l'available togliendo le risorse richieste (per la riga i-esima), mentre per l'allocation le sommo! (infine aggiusto il need)

|       | <u>Allocation</u> | <u>Need</u>  | <u>Available</u> |
|-------|-------------------|--------------|------------------|
|       | A B C             | A B C        | A B C            |
| $P_0$ | 0 1 0             | 7 4 3        | <b>2 3 0</b>     |
| $P_1$ | <b>3 0 2</b>      | <b>0 2 0</b> |                  |
| $P_2$ | 3 0 2             | 6 0 0        |                  |
| $P_3$ | 2 1 1             | 0 1 1        |                  |
| $P_4$ | 0 0 2             | 4 3 1        |                  |

→ modifiche *virtuali* per capire se possiamo cadere in una situazione di stallo o meno!

(ovvero se lo stato è sicuro o no)

→ si riusa quindi l'algoritmo del banchiere (parte della verifica) per capire se lo stato è sicuro  
→ riaprovo l'algoritmo → dunque work diventa pari a l'available, dunque [2,3,0] →  
sicuramente il need è soddisfatto → per cui aggiungo al work le risorse allocate e a quel  
punto cerco un altro processo che posso soddisfare...

• Work = [2 3 0] →  $A_1 \rightarrow [5 3 2] \rightarrow A_3 \rightarrow [7 4 3] \rightarrow A_4 \rightarrow [7 4 5] \rightarrow A_0 \rightarrow [7 5 5] \rightarrow A_2 \rightarrow [10 5 7]$

→ per cui anche stavolta arrivo ad uno stato sicuro tramite la sequenza <P1,P3,P4,P0,P2>  
(come al solito una fra le tante che può essere seguita)

+Si aggiorna quindi effettivamente lo stato a quello nuovo... a questo punto segue una  
nuova richiesta → P4(3,3,0) → la quale è una richiesta valida in quanto è inferiore al need,  
ma non supera il test delle risorse disponibili! → infatti delle risorse A ce ne sono disponibili  
solo 2... per cui si aspetta

→ se invece facciamo la richiesta P0(0,2,0) → anche questa è una richiesta che ha senso  
per lui, e inoltre sono disponibili anche le risorse! → ma se andiamo poi a vedere con  
l'algoritmo di verifiche le nuove risorse disponibili diventano [2,1,0] che sono tutte minori di  
qualsiasi need che possiamo soddisfare! → questo dunque porta ad una situazione NON  
SICURA → per cui la risorsa richiesta da P0 non viene concessa! anche se è disponibile!  
→ magari aspetta che qualcuno termini e lasci le risorse... rallento il sistema! → l'algoritmo  
infatti ferma il processo su qualcosa che lui potrebbe fare ma che potrebbe (non per forza)  
portare a situazione di stallo! (perdita di performance...)

## Rilevamento dello stallo

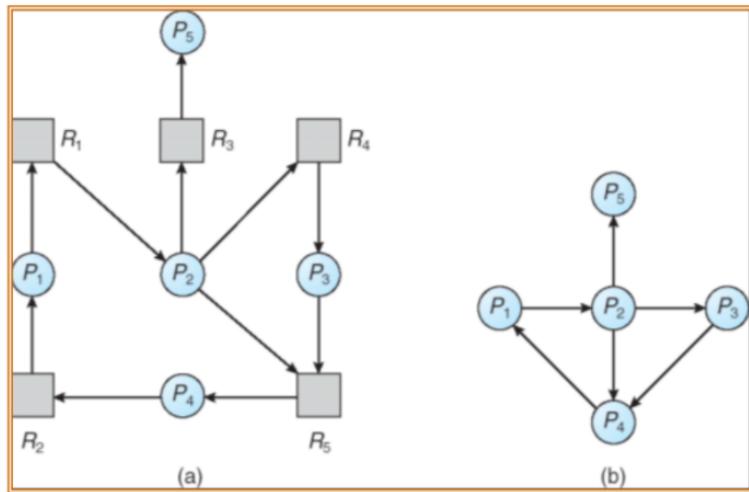
In questo caso si **permette** al sistema operativo di entrare in stallo, per cui il sistema operativo controlla se c'è un processo che si trova in una situazione di stallo... ovvero che aspetta risorse possedute da altri e abbiamo una situazione di blocco  
→ una volta stabilita la presenza di uno stallo si dovrà cercare di risolverlo! → in particolare usare lo schema di ripristino per terminare questa situazione → ad esempio **uccidendo** un processo - fermarlo - farlo ripartire (ad esempio con un rollback) ecc...

### Caso a singola istanza

+ Supponiamo quindi di essere nel caso di una singola istanza per ogni tipo di risorsa → si usa un *grafo di attesa* → dove stavolta sono presenti pure dei legami fra processi! → per dire che un processo sta aspettando un altro processo (ad esempio per sottolineare il caso in cui 2 processi condividono una stessa risorsa per cui un processo aspetta l'altro)  
→ per cui una volta costruito il grafo di attesa → con vertici che sono processi e gli archi che indicano se un processo è in attesa di un altro → nel caso della presenza di un ciclo → il sistema potrebbe avere uno o più processi in stallo!  
→ ovviamente c'è la possibilità di avere più cicli... e dunque cercare di risolverli tutti...

→ costo dell'algoritmo per cercare la presenza di un ciclo nel grafo è di  $O(n^2)$  con n il numero di vertici del grafo!

+Esempio:



→ si passa da un grafo di allocazione delle risorse all grafo di attesa! → per esempio ho  $P_1$  che richiede  $R_1$ , ma questo è posseduto da  $P_2$  → per cui  $P_1$  risulta in attesa! → e da qui l'arco che collega  $P_1$  a  $P_2$  nel grafo, stesso discorso per gli altri processi...

→ si nota quindi nel grafo d'attesa la presenza di cicli → ad esempio fra  $P_1, P_2, P_3$  e  $P_4$ ; oppure quello fra  $P_1, P_2$  e  $P_4$  → il sistema dunque stabilisce che c'è uno stallo fra quei 4 processi...

se però nella risoluzione dello stallo si decidesse di far terminare solo  $P_3$  → in realtà non si risolve il problema! → è presente infatti un altro ciclo... nel caso della scelta dei processi dovremo stare attenti se abbiamo doppi cicli o cicli *interni*...

## Caso più istanze per ogni risorsa

Cosa succede nel caso di più istanze per ogni risorsa? → si usa un algoritmo che opera in modo simile all'algoritmo del banchiere anche se hanno diverse funzionalità → il primo infatti opera per **prevenire** lo stall! → dunque da non confondere... (due situazioni diverse)  
+Abbiamo un array **available** → Un vettore di lunghezza m indica il numero di risorse disponibili per ogni tipo di risorsa.

→ una matrice **Allocation** → indica il numero di risorse allocate sui singoli processi  
→ infine abbiamo una matrice delle **Request** → Una matrice n x m che indica la richiesta corrente di ogni processo. Se Request [ i ][ j ] = k, allora il processo Pi sta richiedendo k risorse di un certo tipo della risorsa Rj  
→ come al solito si fà una *fotografia* del sistema e si valuta se siamo in una situazione di stall o meno! → si vuole controllare se si riesce a finire o meno!

Algoritmo di rilevazione dello stallo

Anche stavolta si usa un array di work e un array di finish → sempre rispettivamente di lunghezza m(numero delle risorse) e n(numero dei processi)

1. Si imposta inizialmente work = available e Finish(i) a false nel caso Allocation(i) risulti diverso da zero! → altrimenti a true → in modo da indicare che il processo può terminare
2. Si cerca quindi un processo che abbia ancora Finish a falso (ovvero non si può dire se riesce a terminare o meno) e che abbia Request(i) <= Work! → se le richieste che in quel momento fanno possono essere risolte usando quelle disponibili... se non viene trovato nessun i allora si salta alla fine (passo 4)
3. Trovato un certo i, per cui uno di quei processi abbia delle richieste che possono essere risolte allora si prende il vettore di work e aggiungiamo le allocazioni possedute dal processo i-esimo → work = work + Allocation(i) → si pone Finish[i] = true e si torna al passo 2 alla ricerca di un nuovo processo che può terminare → stavolta con il vettore di work diverso da quello iniziale!
4. Si arriva quindi infondo nel caso o tutti i processi abbiano finish a true → dunque il sistema non è in stall, oppure nel caso al passo 2 non si riesca a trovare nessun processo terminabile allora non siamo in uno stato sicuro e in particolare i processi con Finish(i) == false allora **Pi è in stall!**

→ L'Algoritmo richiede un ordine di  $O(m \times n^2)$  operazioni per rilevare la condizione di stallo!  
→ per migliaia di processi diventa un sacco costoso... dunque non è da eseguire ogni secondo in quanto ci mette un certo tempo!

+Esempio:

- Cinque processi  $P_0 \dots P_4$ ; tre tipi di risorse A (7 istanze), B (2 istanze), and C (6 istanze).
- Snapshot a tempo  $T_0$ :

|       | <u>Allocation</u> | <u>Request</u> | <u>Available</u> |
|-------|-------------------|----------------|------------------|
|       | A B C             | A B C          | A B C            |
| $P_0$ | 0 1 0             | 0 0 0          | 0 0 0            |
| $P_1$ | 2 0 0             | 2 0 2          |                  |
| $P_2$ | 3 0 3             | 0 0 0          |                  |
| $P_3$ | 2 1 1             | 1 0 0          |                  |
| $P_4$ | 0 0 2             | 0 0 2          |                  |

→ notare che non c'è più la matrice dei max... in questo esempio non sono disponibili alcune risorse → che dunque risultano occupate! (somma delle allocazione delle risorse danno il numero di istanze...)

→ Sarò in stallo?? Noto subito che  $P_0$  non richiede nessuna risorsa e dunque può terminare! → stesso discorso per  $P_2$  → si liberano quindi [3 1 3] risorse con le quali riuscirò a sbloccare il resto dei processi! → dunque con la sequenza  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  porterà ad avere  $\text{Finish}[i] = \text{true}$  per ogni  $i$  → per cui non sono in una situazione di stallo! → tutti potenzialmente riescono a terminare!

+Mettiamo invece che  $P_2$  richieda una risorsa in più...

|       | <u>Allocation</u> | <u>Request</u> | <u>Available</u> |
|-------|-------------------|----------------|------------------|
|       | A B C             | A B C          | A B C            |
| $P_0$ | 0 1 0             | 0 0 0          | 0 0 0            |
| $P_1$ | 2 0 0             | 2 0 2          |                  |
| $P_2$ | 3 0 3             | 0 0 1          |                  |
| $P_3$ | 2 1 1             | 1 0 0          |                  |
| $P_4$ | 0 0 2             | 0 0 2          |                  |

→ si rilasciano le risorse solo del processo  $P_0$  che non ci permette poi di rilasciare alcun altro processo! → siamo in stallo! → in particolare lo stallo risulta essere formato dai processi  $P_1, P_2, P_3$  e  $P_4$

+Quando eseguire l'algoritmo di rilevamento? → dipende da vari parametri... per cui bisogna chiedersi ogni quando lo stallo potrebbe verificarsi?? esistono situazioni in cui si verificano degli stalli di frequente o no?? ( a seconda di processi CPU-intensive o meno...) → Poi dipende anche da quanti processi dovranno essere annullati... ovvero fare il rollback dei processi che non stanno finendo e se ne deve fare uno per ogni ciclo disgiunto (a seconda poi della presenza o meno di cicli sovrapposti) → Se l'alg. di rilevamento è invocato arbitrariamente, possono formarsi molti cicli nel grafo delle risorse ed è difficile determinare quale dei processi in stallo ha "causato" lo stallo.

abbiamo infatti visto nel caso di cicli sovrapposti a seconda del processo che fermo, il ciclo può ancora rimanere o meno!! → dunque devo essere in grado di scegliere il processo da terminare che mi spezzi il ciclo!

→ inoltre va considerato anche il suo costo computazionale abbastanza elevato! → per un miglialo di processi avrò un miglio da lavorare...

+Il ripristino dallo stallo consiste dunque o nel **terminare** i processi, oppure togliere le risorse facendo al **prelazione**

#### *Terminazione dei processi*

Per ripristinare la situazione dello stallo si possono attuare diverse strategie:

- Terminare tutti i processi in stallo → *tabula rasa*
  - Terminare un processo alla volta fino all'eliminazione del ciclo
- in base a cosa scegliere il processo da terminare??
- Nel caso di più processi mi posso basare sulla priorità dei processi → quelli a più bassa priorità i più probabili ad essere terminati
  - Per quanto tempo ha lavorato un processo e per quanto tempo rimane(se però ho queste informazioni...) → conviene buttare via quei processi che hanno iniziato da poco rispetto a quelli che già hanno incominciato da tanto tempo e quindi dovrei buttar via tanto tempo di CPU!
  - Basarsi sulla risorse che il processo ha usato oppure quelle necessarie al completamento (se le conosciamo)
  - Numero di processi che devono essere terminati
  - Il tipo di processo... se interattivo o batch! → per quest'ultimi saltarli una volta  
*"potrebbe non essere una tragedia"*

#### *Prelazione delle risorse*

- **Selezionare una vittima** – minimizzando il costo. (es. n. risorse possedute, quantità di tempo già spesa)
- **Fare Rollback** – ritornare ad uno stato sicuro e far ripartire il processo da questo stato. (tipico delle transazioni nelle basi di dati)
- **Attesa indefinita** – diversamente dal primo caso uno stesso processo selezionato sempre come vittima → per cui questo può soffrire di una attesa indefinita, in quanto non è mai concluso! → per cui si potrebbe includere il numero di rollback nel fattore di costo → in modo che ad un certo punto si potrebbe scegliere un altro processo come vittima anche se di priorità maggiore! (nel momento di selezione di una vittima)

# 19/05/2021

## Gestione della memoria

Back to calcolatori...

Quali compiti ha il sistema operativo rispetto alla memoria??

1. Assegnare ad ogni processo la memoria di cui ha bisogno per la sua esecuzione → compito del sistema operativo sarà infatti quello di caricare in memoria l'eseguibile

per far partire il nostro calcolatore → verrà quindi creato questo processo al quale verrà caricata l'istruzione e la parte di inizializzazione presente nei file eseguibili → quindi viene caricato in memoria e in più vengono riservate delle zone di memoria per tenere i dati che il processo utilizza

2. **Isolare** i processi facendo in modo che i processi non possano accidentalmente o in modo malevolo accedere o modificare zone di memoria riservate ad altri processi o al sistema operativo stesso → parte della memoria è infatti riservata al sistema operativo per l'esecuzione di tutti i processi ed altri aspetti del sistema
3. Utilizzare la memoria nel modo più efficiente possibile in modo da poter aumentare il grado di multiprogrammazione → ovvero quanti processi riescono ad essere eseguiti sul sistema! (con ragionevole efficienza) → essendo infatti finita, questa non può ospitare più di un certo numero di processi...
4. Permettere quando questo sia richiesto esplicitamente la condivisione di zone di memoria tra i processi → ad esempio nel caso avessimo in esecuzione più processi dello stesso tipo (da parte anche di più utenti) → il codice essendo lo stesso può essere condiviso tra tutti i processi che usano quel codice → **risparmiando** così memoria nel condividere uno stesso codice fra più processi!

## Swapping

Primo modo per gestire l'uso della memoria! → quando non c'è più memoria... che succede?? → Ad esempio preso un sistema che deve eseguire più processi contemporaneamente (fino a che può) → dovrà quindi decidere quali possono stare in memoria e quali no!

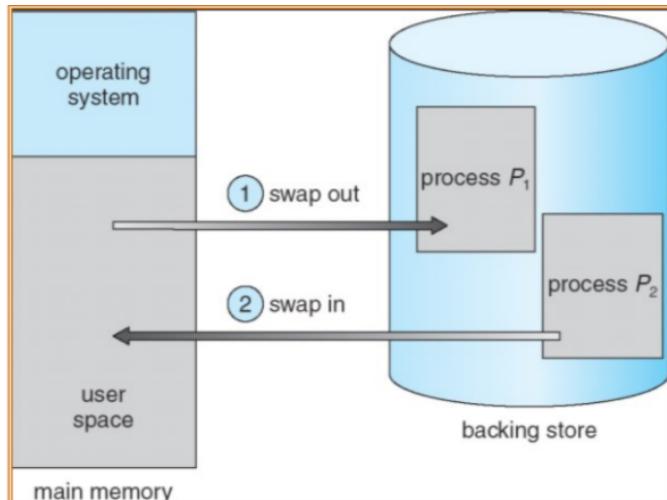
→ Dunque in certi momenti, a seconda di varie politiche, posso fare lo swap di certi processi → ovvero tolgo un processo dalla memoria e lo metto su un cosiddetto *Backing store* → dispositivo di memorizzazione (secondario) su un disco → che dunque riesce a salvare molti dati ma è molto lento!

→ lo stato dei processi passa quindi dalla memoria principale allo *Backing store*... per poter essere quindi ritirati in sù, quando ci sarà di nuovo disponibilità di memoria!

→ Abbiamo dunque il così detto *Roll in - Roll out* → processi che entrano ed escono dall'esecuzione → di solito sono i processi a più bassa priorità ad essere swapped sul disco → in modo che i processi a più alta priorità poi possono essere caricati ed eseguiti!

→ La maggior parte del tempo è dunque occupata dal trasferimento dalla memoria al disco e viceversa... proporzionale alla quantità di memoria trasferita → infatti si cerca di trasferire solo i dati e non il codice → soprattutto il fatto di riscrivere 2 volte non ha molto senso per cui uno lo rilegge da dove era stato caricato!

+Ancora oggi in alcuni sistemi operativi di UNIX, linux o windows è presente lo swapping...



→ prima swap out poi l'in → per gestire la possibilità di avere in esecuzione nella stessa memoria più processi!

#### Spazio indirizzi logico vs fisico

Tutte le tecniche di gestione della memoria si basano sulla divisione dello spazio su indirizzi logici e indirizzi fisici! → l'indirizzo logico è quello generato dalla CPU → quindi dal programma in esecuzione (ovvero l'applicazione) → indirizzi che servono sia per accedere ai dati sia al codice → che poi verranno trasformati in indirizzi fisici della memoria disponibile sul sistema! (indirizzo logico anche virtuale perchè potrebbe non essere esplicitamente uguale a quello fisico)

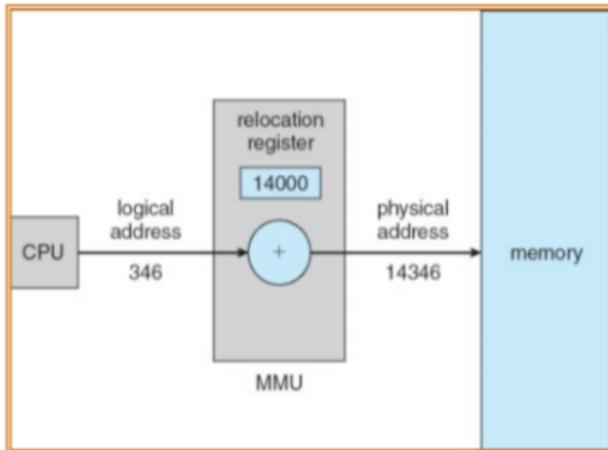
→ Si ha che ogni processo ha il proprio spazio di indirizzi logici! → ogni processo quando si riferisce ad una certa zona di memoria questa è propria del processo e distinta da altre zone di altri processi!

→ anche se si riferiscono allo stesso indirizzo poi in realtà fisicamente faranno riferimento a zone fisiche della memoria diverse... in modo che questi non si sovrappongono (nel caso non sia richiesto esplicitamente) → in generale quindi i processi ognuno ha il suo spazio degli indirizzi → quindi vive di vita propria e non può interagire con altri spazi per altri processi!  
→ ogni processo risulta isolato dagli altri! → dunque non può nemmeno involontariamente accedere a spazi di memoria degli altri processi!

+Più spazi di indirizzi logici poi sono mappati sullo stesso spazio di indirizzi fisico, in quanto la memoria non è infinitivamente estesa... da A[0], A[1]... fino all'estensione della memoria

#### Memory management unit (MMU)

+Per fare la traduzione tra gli indirizzi logici e quelli fisici, in realtà è presente un dispositivo detto *MMU* (tempo fa al di fuori della CPU, oggi invece è integrato) che permette di mappare gli indirizzi logici in indirizzi fisici!



→ La CPU produce un indirizzo logico del processo che ha in esecuzione → questo passa all'MMU il quale produce un certo indirizzo fisico che andrà quindi a finire alla memoria, quindi alla parte del sistema che si occupa della gestione della memoria!

+Nell'esempio l'MMU si basa un *registro di rilocazione* che dice da dove inizia lo spazio di memoria del processo di esecuzione... prende l'indirizzo logico, lo somma al contenuto del registro di rilocazione e quindi genera l'indirizzo fisico! → più semplice MMU che si possa fare! → per questo tipo di approccio si parla di *rilocazione dinamica*!

### Tecniche di gestione della memoria

+Anche questa una tecnica dunque semplice... Altre tecniche per la gestione della memoria sono:

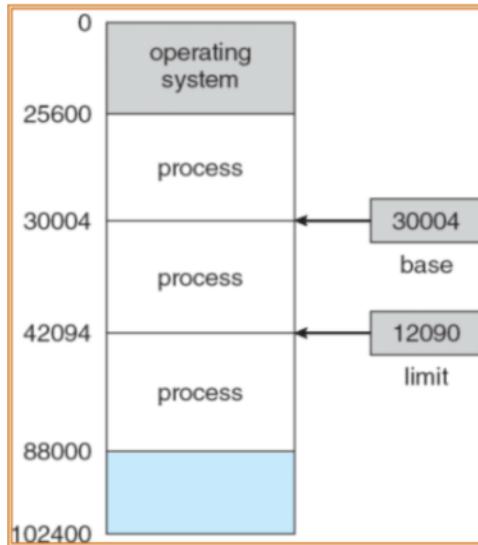
- Allocazione contigua
- Segmentazione (vista con CC) → sistema alloca vari segmenti in zone diverse della memoria!
- Paginazione (la più usata... risolve dei problemi dell'allocazione contigua e la segmentazione)
- Segmentazione paginata → che integra le due → sia per sfruttare la segmentazione che per essere retroattivi con gli hardware precedenti

### Allocazione contigua

→ La memoria viene divisa in 2 parti:

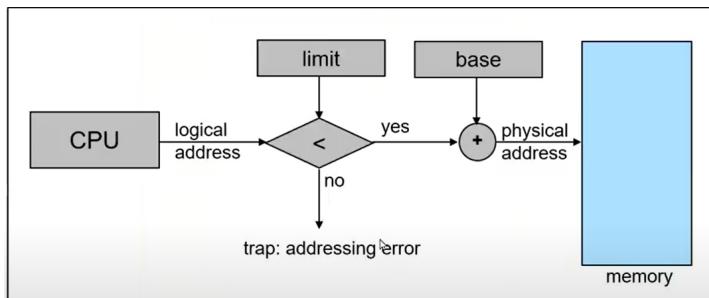
- Una con codice e dati del sistema operativo → tenuta di solito nella parte bassa, insieme al vettore delle interruzioni → sistema operativo di solito infatti viene messo dove sta il vettore delle interruzioni (alcuni processori lo hanno ai primi indirizzi altri sugli ultimi...)
- L'altra parte (ciò che rimane) viene usata per tenere i vari processi degli utenti...

+Ogni processo ha associato una sola partizione (data dal sistema operativo) → una zona di memoria che lui ha a disposizione → il suo indirizzo logico parte dall'indirizzo zero e arriva alla dimensione massima della partizione che gli è stata associata!



→ il processo ha un **indirizzo di base** che gli è stato associato, e poi c'è un **valore di limite** che è la dimensione! → la rilocazione viene fatta basandosi su queste 2 informazioni → da dove inizia il processo e qual è la sua dimensione massima! → si deve infatti evitare che un processo accedi a zone di memoria del processo successivo (indirizzi negativi non esistono... per cui al massimo andrà al successivo) → si devono controllare che i riferimenti siano validi ovvero entro il limite che gli è stato posto!  
 → la somma fra base e limite mi dà appunto la dimensione riservata al processo! → limite intenso come dimensione massima raggiungibile

+Come funziona quindi questa MMU? (nel caso di allocazione contigua)



→ dalla CPU esce l'indirizzo logico, lo compara con il valore del limite (contenuto di un registro) → se è minore ok e si va avanti, se invece non è così ovvero si sta tentando di andare oltre il limite che è stato assegnato al processo che è in esecuzione sulla CPU, viene quindi generata una eccezione → da gestire fermando il programma ed entrando in esecuzione il sistema operativo

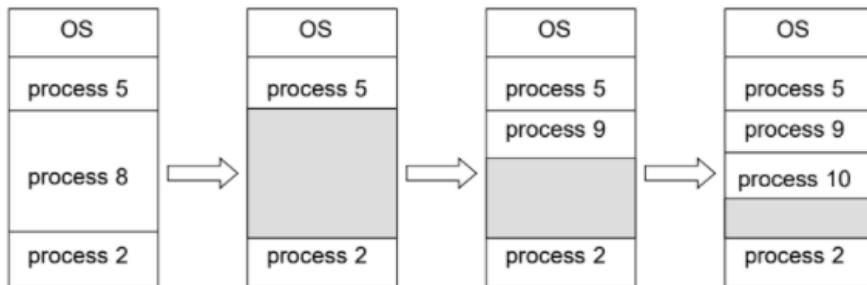
→ tornando al caso in cui non si lancia un'eccezione, si somma l'indirizzo logico con una base → generando quindi l'indirizzo fisico, pronto per essere letto dalla memoria

+Quando dunque viene fatto il context switch di un altro processo significa che oltre a cambiare il processo, si cambia la regione di memoria dove lui può accedere → per cui quando viene eseguito una delle informazioni che deve essere reimpostata durante il context switch è quella di reimpostare i valori dei registri base e limit → in modo che il processo successivo a cui viene assegnata la CPU si riferisce alla zona di memoria di un determinato processo e non quella di altri!

→ compito del sistema operativo, sarà quello di tenere "aggiornata" la MMU attraverso i suoi registri per andare ad usare la giusta zona di memoria affidata al processo! (soprattutto nel caso di caso di context switch → dove i valori da aggiornare saranno presi dal **PCB** → il quale ricordo tiene tutte le informazioni relative al controllo di un processo)

+Cosa succede invece nel caso di allocazione di più processi??

→ Abbiamo delle zone di memoria assegnate ai vari processi...



→ Ad un certo punto il processo 8, termina → dunque libera la zona di memoria che gli era stata assegnata! → si un cosiddetto *buco* nella memoria che possiamo usare per soddisfare tutte le allocazioni dei vari processi

→ nel momento in cui bisogna affidare della memoria ad un processo, viene cercato un buco abbastanza capiente per tenere la richiesta che viene fatta dal processo che deve essere messo in esecuzione

→ il sistema operativo dovrà quindi tenere informazioni su quali sono le partizioni allocate → ovvero dove sono e quanto sono grandi, queste zone → però devono avere anche informazione su dove sono i buchi! → un lista che sarà poi utile per soddisfare le varie richieste di allocazioni di memoria

+Supponiamo però il caso in cui il processo 9, abbia terminato dunque lascia un buco → arriva un nuovo processo che ha bisogno delle dimensioni lasciate dal processo 9 che quelle che c'erano già... la richiesta non viene accettata! → fallisce oppure viene messo in attesa che si liberino altre zone di memoria! → questo problema è detto *problema della frammentazione*

+Come soddisfare la richiesta di allocazione? → abbiamo diverse strategie:

- **First-fit:** Alloca il **primo** buco abbastanza grande per tenere la richiesta
- **Best-fit:** Alloca il buco che lascia "meno scarto fra tutte le possibilità" → tra tutti i buchi che possono soddisfare la richiesta... sceglie quello che vi si avvicina di più! → che quindi produce il buco libero più piccolo! (può capitare quindi di scorrere tutti i buchi oppure doverli ordinare...)
- **Worst-fit:** cerca quello più grande! → in modo da poter generare il buco libero più grande fra tutti quelli possibili! → in modo che sia probabile che il buco sia riutilizzabile...

→ dopo varie misurazioni si osservato che First-fit e best-fit sono migliori di worst-fit in termini di velocità e utilizzazione di memoria

### Problema della frammentazione

→ La **frammentazione esterna** è quella per cui esiste uno spazio di memoria per soddisfare la richiesta che deve essere fatta dal nuovo processo che deve essere allocato però non è contigua! → ci sono tante zone di memoria (piccole) le quali sommate potrebbero

soddisfare la richiesta fatta ma singolarmente non ce n'è nessuna che possa soddisfare la richiesta! → il processo non può essere messo in esecuzione e quindi dovrà aspettare!

+Oppure si attua una strategia di *compattazione* → si sposta tutte le zone di memoria dei processi attualmente in esecuzione in modo da raggruppare gli spazi liberi... è però questa un'attività molto costosa a livello di sistema e in particolare blocca l'esecuzione dei processi che sono in esecuzione durante questo spostamento → fermando l'esecuzione per un tempo non trascurabile... dunque si cerca di evitare questo tipo di approccio!

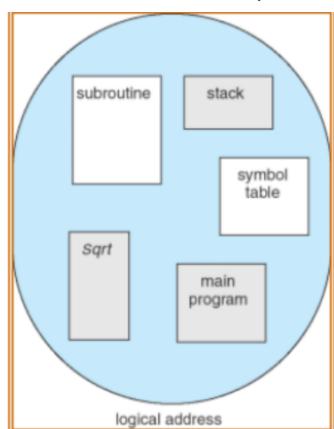
+Infine esiste un altro tipo di frammentazione → quella **interna** → la memoria allocata ad un processo può essere leggermente superiore a quella richiesta dal processo! → gli viene data la possibilità di accedere ad una zona di memoria che in realtà il processo non usa!

## Segmentazione

+A meno di sistemi ormai obsoleti oppure alcuni sistemi embedded, l'allocazione contigua non è più usata → la gestione della memoria si è quindi evoluta nella tecnica di segmentazione!

→Dove in realtà non si fà altro che usare l'allocazione contigua ma dividendo i vari spazi a cui il nostro processo può accedere...

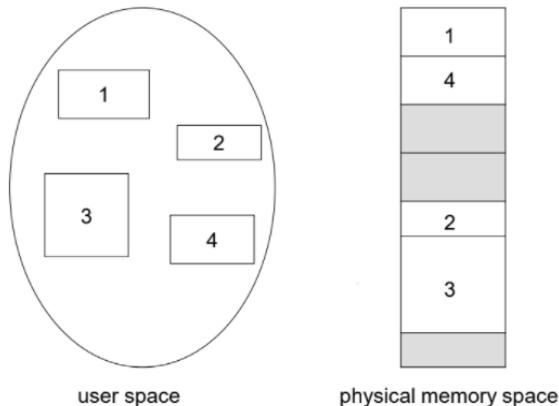
→ un programma infatti non ha bisogno che tutta la memoria sia contigua → ma solo alcune parti ne hanno bisogno! → ad esempio per una funzione(come una procedura) di un programma ho magari la necessità che le istruzioni di quella istruzione sia contigue fra di loro e oltre l'istruzione return viene meno questo bisogno! → conviene comunque mettere in parti separate le varie procedure di un programma dove per ogni procedura le istruzioni siano vicine fra loro, ma le varie procedure possono stare anche lontano fra loro



+Per la gestione dei dati invece, ad esempio gli array → potrei mettere questi in un segmento, oppure avere le variabili locali/globali → lo stack separato da... ecc

→ il nostro processo può essere quindi visto come un insieme di segmenti sui quali il posizionamento reciproco non è definito!

Ad esempio:



→ di solito il numero dei segmenti non è molto elevato → potremo avere il segmento del codice, dei dati, dello stack/heap (se lo vogliamo separare) → e quindi sono in numero limitato → i quali saranno comunque mappati nella memoria fisica nel modo scelto dal sistema operativo, in base alle disponibilità di memoria...

## Architettura

+Gli indirizzi logici che vengono forniti dalla CPU sono in realtà, fatti da 2 valori  
 → il numero del segmento e l'offset! → quindi abbiamo la **tabella dei segmenti** → la quale associa ad ogni segmento: una base e un limite!  
 → in particolare è presente un registro di base della tabella dei segmenti detto *Segment-table base register* – STBR → che punta alla tabella dei segmenti dei processi in esecuzione!

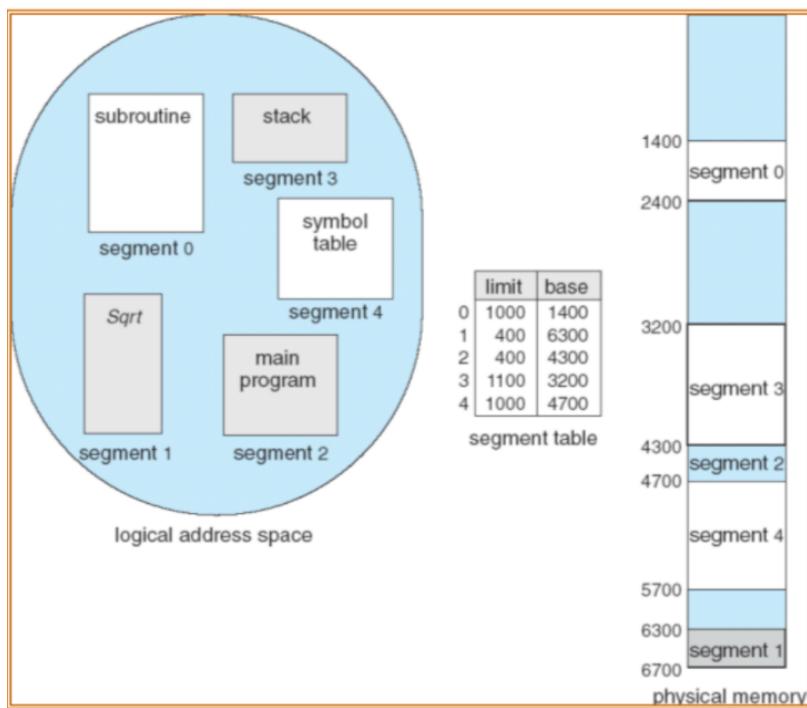
+Dunque questo tipo di gestione della memoria usa:

- la **rilokazione** → dinamica, e si sfrutta la tabella dei segmenti
- la **condivisione** → si ha la possibilità di condividere la memoria tra processi diversi... in quanto basta definire 2 segmenti e fare in modo che questi puntino alla stessa zona di memoria! → posso quindi fare in modo che alcune parti della memoria siano condivise ed altre no → cosa non possibile nella allocazione contigua dove ogni processo deve usare solo la propria zona di memoria assegnata
  - + Se però abbiamo la necessità di condividere allora si deve stare attenti ad usare lo stesso numero di segmento
- la **allocazione** → per ogni segmento si può usare first/best fit → ma anche qui abbiamo il problema della frammentazione esterna! → meglio però del caso precedente in quanto si gestiscono zone di memoria più piccole e quindi risulta meno evidente il problema dei vari blocchi creati inizialmente e fissi → qua la situazione può essere più dinamica!

+Nella tabella dei segmenti dunque oltre all'informazione base e limite è possibile che ci siano anche altri bit che danno le info su di un certo segmento, ad esempio:

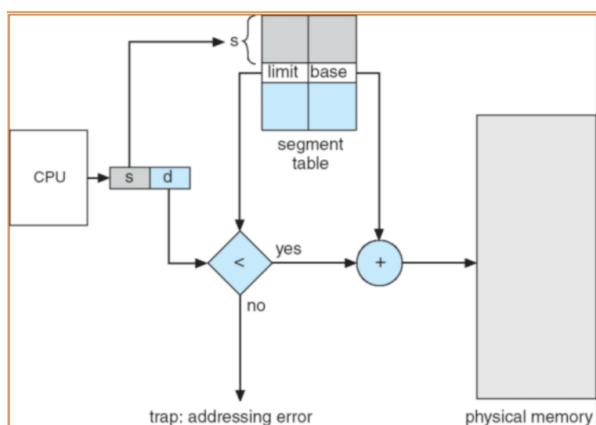
→ Bit di validazione (0 o 1) per dire se un segmento risulta valido o meno  
 → Bit che dicono se i dati letti possono essere letti/scritti oppure se possono essere eseguiti su quel segmento → questi dunque costituiscono dei bit di protezione associati ai segmenti  
 → e come detto prima la condivisione del codice avviene a livello di segmento  
 → infine come detto inizialmente siccome i segmenti avranno lunghezza diversa l'allocazione della memoria è un problema di allocazione dinamica!

+Esempio:



→ calcola e definire le dimensioni dei vari segmenti → ovvero nel momento in cui il sistema operativo fa partire il processo guarda da cosa è composto → alloca i segmenti e costruisce la tabella dei segmenti → dando limite e base per ogni segmento (limite impostato tramite le info che ci sono sull'eseguibile, mentre la base sarà definita dal sistema operativo in base alle disponibilità che sono presenti nel sistema)

+Come funziona l'MMU in this case?

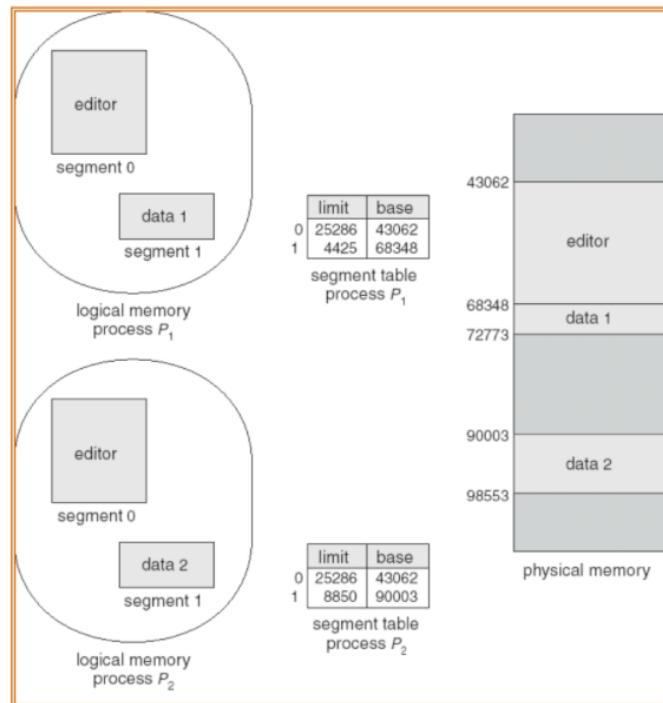


→ come al solito la CPU genera l'indirizzo logico, stavolta però diviso in numero del segmento e offset all'interno del segmento! → sulla base di questo numero si va nella tabella dei segmenti e vedere qual è quel segmento al quale la CPU vuol fare riferimento verificando limite e base!

→ in particolare con il limite si controlla che l'offset dell'indirizzo logico sia valido (ovvero minore del limite → dunque entro lo spazio concesso) → se non è così si lancia la trap (con sfera ebbasta) → errore di indirizzamento → entra il SisOp e decide che fare...

→ stabilito invece che la richiesta sia valida allora si prende la base del segmento... la somma al *displacement* (offset per piersa?) → e quindi genero l'indirizzo fisico per accedere alla memoria!

+Esempio di condivisione dei segmenti:



→ in entrambi i processi il segmento zero è l'editor → ad esempio supponendo il processo P1 parta per primo → alloco lo spazio richiesto e a quel punto il processo P2 richiede la memoria → P1 è già in esecuzione e quindi si decide di riusare la zona di memoria già allocata per l'altro processo! → nella tabella dei segmenti dunque si assegna la stessa base che è stata data a quell'altra tabella!

→ si ha quindi la possibilità di risparmiare la memoria per una seconda istanza dello stesso programma! (sotto l'ipotesi di programmi non automodificanti → altrimenti sta roba non si fà)

+Alcuni Pro/contro (rispetto alla allocazione contigua)

- Molto più flessibile rispetto alla allocazione contigua → si ha anche la possibilità di allargare o spostare i segmenti!
  - Permette condivisione dati/codice tra processi
- contro però c'è il fatto del problema della frammentazione esterna! → si può arrivare a situazioni del sistema in cui non si riesce a trovare un buco abbastanza grande per una delle richieste del programma → processo che non può essere messo in esecuzione (a meno anche qui di non attuare una politica di compattazione ma sempre molto costosa...)

## Paginazione

→ Gestione della memoria più usata al momento! → utilizza uno spazio di indirizzo logico poi rimappato sull'indirizzo fisico, in modo molto libero → si basa sulla gestione di **pagine** → tutte di grandezza fissa → gestione della memoria dunque fatta sull'unità che sono le pagine!

→ le pagine hanno in particolare una dimensione che è una potenza del 2 → vanno da 512 byte a circa 8 kilobyte → pagine divise a loro volta in blocchi di lunghezza fissa → anche la

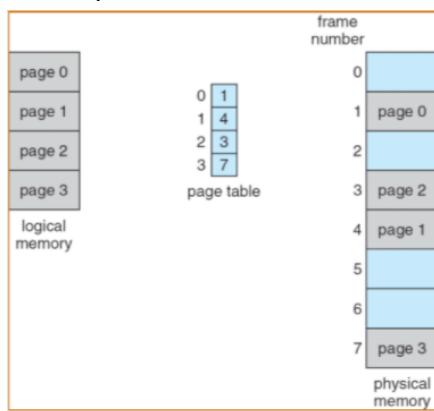
memoria fisica viene divisa in dimensioni dello stesso valore → quando però si parla di memoria fisica si parla di *frame* (!= pagine) → le pagine infatti sono quelle processo (spazio logico), mentre quando si parla di memoria fisica questa è divisa in frame → quello che viene fatto è: associare le pagine ai frame di memoria disponibili!

+Per eseguire un programma che usa  $n$  pagine, il sistema operativo dovrà trovare  $n$  frame liberi e caricare il programma!

→ quindi verrà creata una *tabella delle pagine* che serve a tradurre ogni indirizzo logico in indirizzo fisico → ovvero a dire dove ogni singola pagina del processo dove risulta memorizzata nella memoria fisica! → in particolare quale risulta il frame associato a quella pagina! (ocio → frame e pagina hanno stessa lunghezza → non possono essere diverse!)

+Anche questo tipo di approccio avendo la dimensione delle pagine fisse, è comunque soggetto al problema della frammentazione interna! → se ad esempio un processo ha bisogno di un certo numero di byte, in realtà gli verrà fornito un numero di pagine che supera la dimensione richiesta! → ci sarà comunque un piccolo scarto inutilizzato che dipende dalla pagina presa (in media è metà pagina!)

+Esempio:

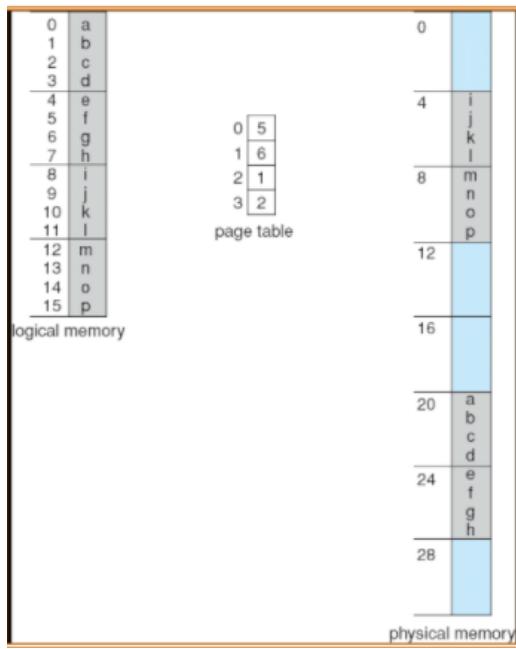


→ a sinistra un processo vede la memoria logica divisa in 4 pagine → a destra invece sono presenti 8 frame → dove dimensione della pagina e del frame è la stessa!

→ al centro è presente la tabella delle pagine che ricordo dice come le pagine vengono associate ai frame → in particolare è un array e ad ogni posizione il numero associato è il numero del frame!

→ si risolve così il problema della frammentazione (a questi punti penso sia quella esterna), dove se ad esempio arriva un processo che richiede 2 pagine, avendone 4 a disposizione posso accettare la richiesta (pagine che non devono essere per forza vicine in memoria! (contigue) → tutto infatti viene definito dalla tabella delle pagine) → questo ovviamente finchè ho frame disponibili...

+Another example:



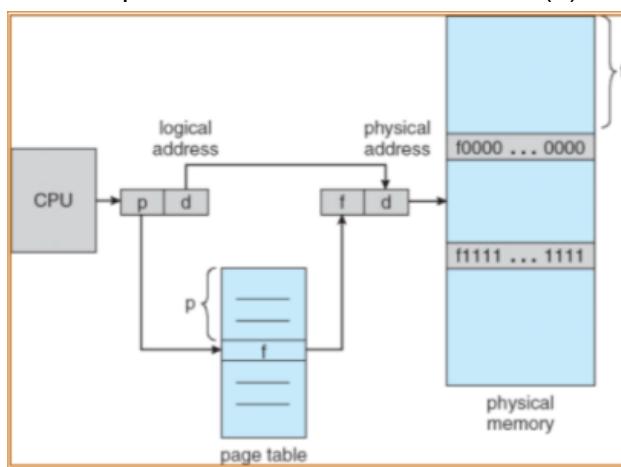
→ 4 pagine ognuna di 4 byte → come al solito pagina 0 messa nel frame 5 (con indirizzo di inizio 20), pagina 1 in 6 e così via...

→ c'è completa libertà su come mettere i dati e associare le pagine con i frame → apparte il fatto che la dimensione dev'essere per forza fissa e non variabile come nel caso della segmentazione!

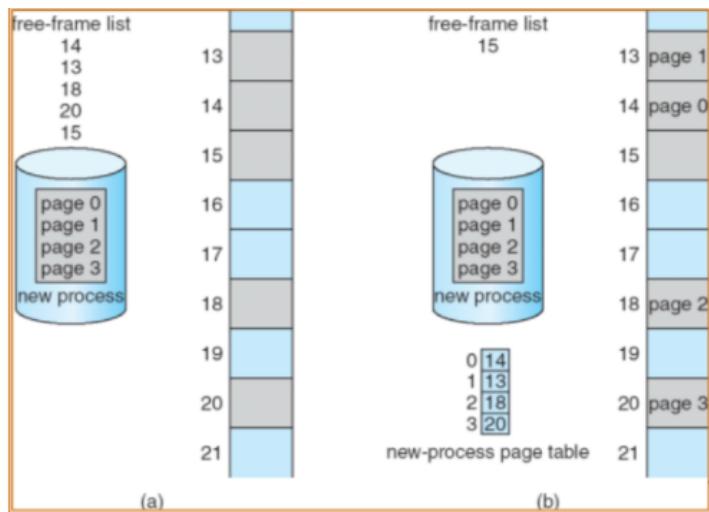
(ocio pt2 → indirizzi in sequenza nella memoria logica non corrispondono a indirizzi in sequenza nella memoria fisica!)

### Schema di traduzione degli indirizzi

L'indirizzo logico è diviso in 2 parti → una parte che rappresenta il *numero della pagina* (p) → e una parte che indica lo *scostamento* (d)



→ l'indirizzo logico composto da numero della pagina e offset (notare che la divisione è possibile solo perchè la dimensione delle pagine è una potenza del 2...) → il numero della pagina viene quindi usato come indice della tabella delle pagine → a quale corrisponde il frame della memoria → quest'ultimo associato allo scostamento all'interno della pagina → insieme costituiscono l'indirizzo fisico!



+Per ogni nuovo processo, il sistema operativo si tiene dei frame liberi → per cui quando ad esempio arriva un nuovo processo, il sistema operativo costruirà la tabella delle pagine per quel processo, allocando nel posto che richiede opportuno! (in grigio sono i frame liberi sta volta...)

### Implementazione tabella delle pagine

La tabella delle pagine è tenuta in memoria → ci sarà quindi una zona in cui il sistema operativo si tiene tutti i dati/informazioni relativi ai processi

→ sarà poi presente un registro di base della tabella delle pagine (registro della CPU) che punta a dove si trova la tabella delle pagine per quel processo che è in esecuzione! (in particolare l'inizio) → del resto delle tabelle non gli interessa!

→ registro detto PTBR → *page-table base register*

+In questo approccio però, per ogni accesso di memoria, ne dovrò fare 2 → devo andare prima ad accedere alla tabella delle pagine per capire quale sia il frame → prendere questo e generare il nuovo indirizzo, per tornare alla memoria e quindi accedere al dato effettivo!  
 → di mezzo dunque le prestazioni del sistema non sono molto efficienti → soprattutto dovuto al fatto di usare la memoria!

### Memoria associativa

+Come risolvere questo problema? Si usa una cache! → viene usata una cache di ricerca veloce detta **memoria associativa** → oppure anche in questo contesto chiamata **TLB** (translation Look-aside Buffer)

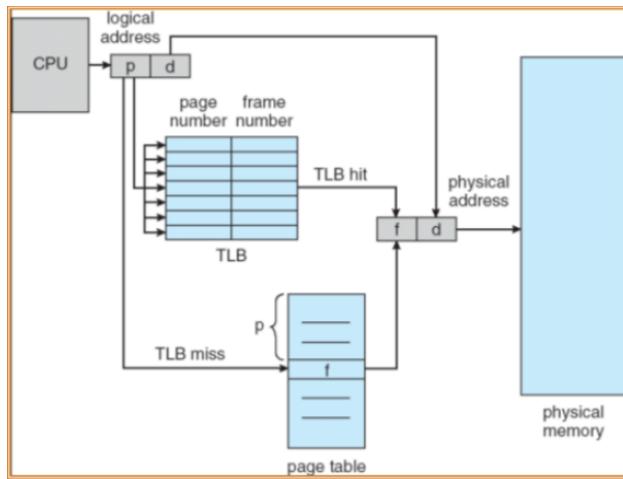
→ memoria associativa nella quale viene fatta una ricerca parallela → dove sono memorizzate il numero delle pagine e il frame!

| n. pagina | n. frame |
|-----------|----------|
|           |          |
|           |          |
|           |          |
|           |          |

→ Dato quindi un valore A' da cercare nella colonna n.pagina, se ne trova il frame associato!

→ tabella però di dimensioni ridotte che può non contenere tutto il contenuto della tabella delle pagine di un processo → ma ne terrà una parte! → dunque nel caso in cui faccio una ricerca e non la trova... si va in memoria!

→ sistema di accesso in memoria diventa diviso in 2 fasi:



→ prima viene controllato se la pagina richiesta è presente nella TLB → in particolare si guarda se è già stato utilizzato il frame associato! → se è presente evito di andare in memoria e dunque faccio presto! → prendo il frame associato, aggiungo l'offset e genero l'indirizzo fisico...

→ se non lo trovo → vado a fare la richiesta nella tabella delle pagine → dunque vado in memoria, trovo il frame e in più memorizzo nella TLB l'associazione trovata → in modo che un prossimo riferimento ad un dato della stessa pagina utilizzi direttamente la TLB invece che andare in memoria!

→ si sfrutta il fatto che spesso i riferimenti alla memoria sono in sequenza → se eseguo un'istruzione è molto probabile che eseguo l'istruzione successiva! → dunque sfruttando la TLB si riesce spesso (non sempre) ad evitare il doppio accesso in memoria!

Tempo di accesso

Ipotizzando che:

- La ricerca associativa utilizzi un tempo epsilon (abbastanza basso)
- il ciclo di memoria duri 1 microsecondo
- Abbiamo un certo tasso di successo (*hit ratio*) → percentuale delle volte che un numero di pagina è trovato nei registri associativi → tasso che dipende dal numero di registri associativi presenti... e questo tasso di successo sia pari ad alpha

→ allora il **tempo effettivo di accesso (EAT)** lo si calcola come:

$$\begin{aligned} EAT &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

→ dove 1 abbiamo detto è il tempo di accesso alla memoria e epsilon quello alla TLB → moltiplicato per la percentuale delle volte che ho successo! → il resto dei casi invece farò 2 accessi in memoria e un accesso alla TLB per modificare il contenuto di una riga... il tutto moltiplicato per il complementare del tasso di successo (numero statistico del tasso di fallimento) → preso quindi:

- $\alpha = 80\% = 0,8$
- $\varepsilon = 0,2$  microsecondi
- EAT =  $2 + 0,2 - 0,8 = 1,4$  microsecondi

→ notare che comunque il tempo di accesso alla memoria viene aumentato! → dovuto al fatto che ho un tempo di accesso alla TLB... comunque questo 40% ce lo tieniamo → ci evita poi il problema della frammentazione...

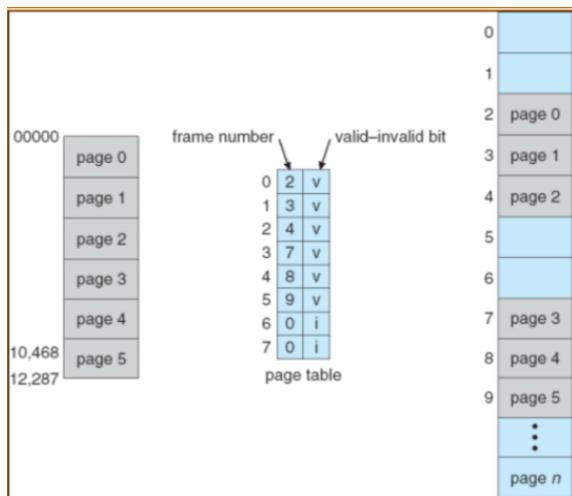
+Inoltre aumentando alpha al 98% ad esempio → si ottiene EAT =  $2 + 0,2 - 0,98 = 1,22$  microsecondi (quindi un aumento minore del 22%)

### Protezione della memoria

In realtà nella tabella delle pagine non è memorizzato solo il frame, ma è memorizzato anche il *bit di validità* → che ci dice se quella associazione fra pagina e frame sia valida o meno! → in particolare nel caso non sia valida ci dice che quella pagina non si trova nello spazio degli indirizzi logici del processore...

→ dunque se il nostro processo va ad accedere a queste zone di memoria del processo stesso che non sono mappate → verrà generata una posizione non valida e quindi sarà generata un'eccezione!

+Esempio:



→ in questo caso già con la pagina 5 ricopriro tutti gli indirizzi logici possibili, per cui nella tabella delle pagine mapperò solo una parte e per la parte finale dirò che quel bit non è valido!

→ al massimo l'indirizzamento logico è fatto da 8 pagine, se ne uso solo 6 le altre 2 pagine non sono usate dunque non vi è nemmeno necessità di allocarle!

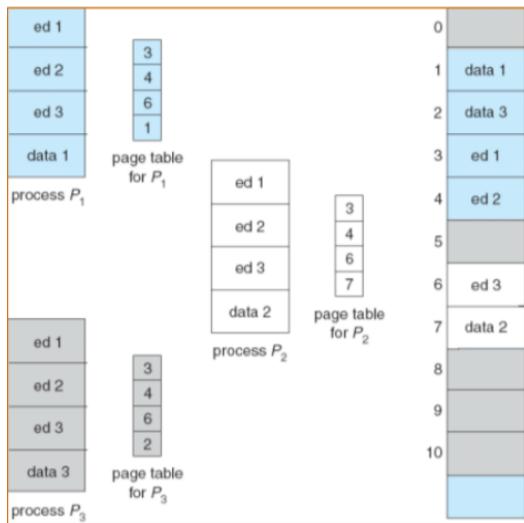
+Risulterebbe inutile associare ad ogni processo tutto lo spazio possibile che lui può allocare! → ma solo le parti che saranno effettivamente usate!

(sfruttando questo fatto inoltre, preso ad esempio un sistema che usa 32 bit per gli indirizzi logici → il tuo processo può generare fino a 4Gbyte → si possono dunque usare molti più indirizzi fisici per mappare più processi! → ognuno dei quali però può indirizzare solo a uno dei 4Gbyte!)

+Con questo tipo di protezione di memoria risulta facile la condivisione delle pagine → si possono condividere fra i vari processi sia per avere del codice condiviso → il quale dovrà essere necessariamente nella stessa posizione dello spazio logico! di tutti i processi!

+In particolare è possibile avere sia una parte condivisa ma anche una privata → in cui ogni processo avrà una copia separata di codice e dati → Le pagine per il codice e dati privati possono essere in qualsiasi posizione nello spazio logico del processo!

+Esempio:



→ i processi  $P_1, P_2$  e  $P_3$  sono sullo stesso programma → per cui hanno lo stesso codice su cui deve essere eseguito, ma operano con dati diversi! → si nota infatti che nelle tabelle delle pagine la parte iniziale mappa negli stessi frame → le stesse pagine dovranno essere condivise fra i 3 processi, mentre i dati saranno ovviamente mappati in frame diversi...

+Esiste anche la possibilità attraverso chiamate di sistema che permettono di allocare zone di sistema che dovranno essere condivise! → allora ci sarà la possibilità di allocare delle pagine che dovranno essere condivise fra vari processi e dunque poter fare cose insieme ("quirky pierfra")

### Struttura tabella delle pagine

Preso ad esempio il processore intel IA32 → che usa gli indirizzi a 32 bit, dove:

- 20 bit sono usati per identificare la pagina →  $2^{20} = 1$  Mega di pagine!
- 12 bit (i rimanenti) usati per l'offset → la dimensione di pagina è quindi  $2^{12} = 4KB$   
→ ogni pagina ha una dimensione fissa di 4Kbyte!
- Ogni elemento della tabella delle pagine usa 4 byte → abbiamo quindi 20 bit per indicare il frame + 1 bit per la validità

→ siccome la tabella delle pagine va allocata completamente → un processo infatti può generare un indirizzo logico valido, ma che è qualsiasi → non esiste come nella segmentazione una dimensione (?) → si dovrebbe quindi mappare tutto lo spazio indirizzabile da quel processo!

→ Dunque se ho 4byte per ogni possibile pagina... avrò 4Mbyte per ogni singolo processo!  
→ il sisOp deve usare per l'esistenza di un processo 4MB e non è semplice gestire tutto lo spazio per tenere un processo di cui usa 100KB + lo spazio che uso per tenere la tabella delle pagine di quei dati effettivamente usati dal processo... il tutto dunque rende questo approccio poco utilizzabile!

(ricordo l'uso del PCB → in particolare nel nostro caso anche la tabella delle pagine sarà contenuta nel PCB → anzi il suo indirizzo → e nel momento in cui si fa context switch si cambierà l'indirizzo della tabella delle pagine...)

+Abbiamo quindi 3 possibilità di risoluzione ai problemi:

1. Paginazione gerarchica → in cui viene costruito un albero
2. Tabella delle pagine di tipo Hash
3. Tabella delle pagine invertita → che ribalta il problema

→ tutto questo andato nel cestino dal momento che oggi si usano indirizzi a 64 bit →

approccio non utilizzabile nel caso di dimensioni troppo grandi!

→ da qui le soluzioni presentate successivamente...

### Paginazione gerarchica

→ Tecnica in cui si divide l'indirizzo logico in tabelle delle pagine multiple → una tecnica semplice è la tabella delle pagine a due livelli... Ad esempio:

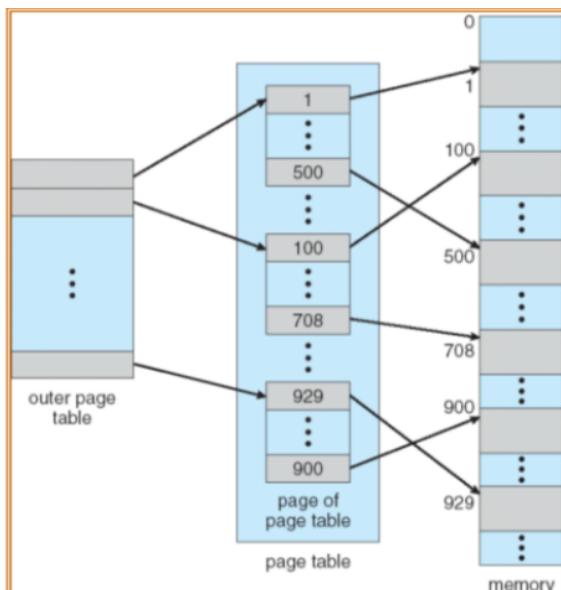
→ l'indirizzo logico a 32 bit viene diviso:

- Numero di pagina a 20 bit
- Offset di 12 bit

→ numero della pagina diviso a sua volta in 2 parti → 10 bit si usano come indice mentre il resto dei bit si usa come indice ma di un'altra tabella

| page number | page offset |     |
|-------------|-------------|-----|
| $p_1$       | $p_2$       | $d$ |
| 10          | 10          | 12  |

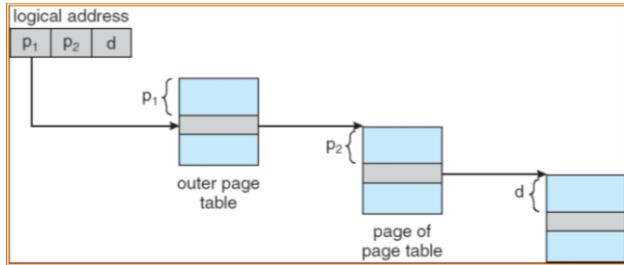
→ dove  $p_1$  è un indice nella tabella delle pagine esterna (*outer page table*), e  $p_2$  è lo scostamento nella tabella delle pagine esterna → vediamo un esempio



→ tramite  $p_1$  seleziono la riga all'interno della outer page table alla quale voglio fare riferimento → mentre  $p_2$  serve ad indirizzare dentro la page table!

→ anche qui si ha la possibilità di organizzazione diversa → dove il vantaggio di questo approccio sta nel fatto di non aver la necessità di costruire tutto l'albero → ma solo la parte che mi interessa! → se ad esempio un processo usa solo la parte in alto non ho neccessità di costruire il resto... si ha una maggiore libertà!

+In pratica cosa succede?



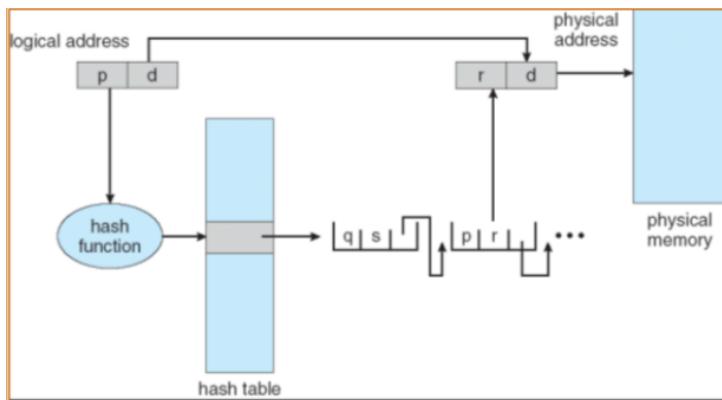
→ p<sub>1</sub> lo uso come indice nella outer page table, ricavo il contenuto indicato dall'indice → tramite questo si determina dove si trova la pagina nella tabelle delle pagine e poi si usa p<sub>2</sub> per sapere quale sia il frame !

+Tecnica che può essere estesa anche a più livelli! → processori a 64 bit usano ad esempio 4 livelli diversi!

### Paginazione con tabelle hash

Abbiamo spesso indirizzi di dimensione maggiore a 32 bit, dove la tabella delle pagine può avere dimensioni incredibili → ad esempio con 64 bit e pagine da  $2^{12} = 4$  Kbyte → avrò 52 bit per numero di pagina → dunque se abbiamo 4 byte per numero di pagina → ovvero avrò  $4 \cdot 2^{52} = 16384$  TByte !! pero ogni tabella delle pagine → cosa un po' inutile...

→ Si usa una tabella hash dunque per associare al numero della pagina il frame!



→ si prende il numero della pagina, lo passa ad una funzione hash che prende e genera tramite la tabella hash un indice → tabella che ha dimensione molto inferiore rispetto a quella possibile → siccome è una funzione hash ci sarà comunque la possibilità di avere collisioni verrà quindi gestita una lista di tutte le pagine che collidono fra di loro! (ovvero allo stesso indirizzo avremmo più associazioni pagina frame! → pagine diverse danno stesso valore hash)

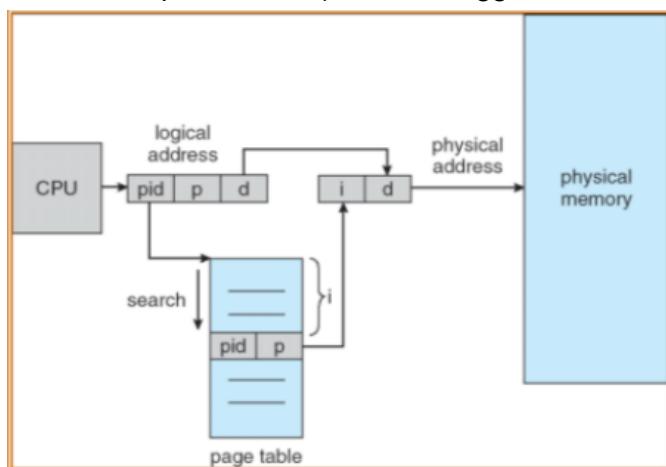
→ in particolare nella lista in figura terrò l'associazione fra la pagina q e il frame s, oppure la pagina p e il frame r, ecc... in media la lunghezza della lista è piccola (questo se la funzione hash è fatta bene!)

→ dunque quando è generato un indirizzo logico, si prende la pagina → genera funzione hash → si prende e si itera sulla lista a cercare l'associazione pagina frame! → nel esempio si ricerca la pagina p dunque si entra nella lista → passo l'elemento q e arrivo alla pagina p! → l'associazione (in questo caso r) verrà aggiunta con l'offset per generare l'indirizzo fisico!

+Anche in questo caso però vengono fatti diversi accessi in memoria → soprattutto per le liste → per ogni accesso infatti non ne abbiamo solo 2, ma può capitare anche 3 o 4 → in questi casi dunque l'uso della TLB è fondamentale!

### Tabella delle pagine invertita

In realtà abbiamo una sola tabella delle pagine per tutti i processi → e questa ci dice per ogni processo, preso un frame questo a quale pagina corrisponde! → la tabella delle pagine non ha tante righe quanto è l'indirizzamento logico → ma quante sono le pagine fisiche... → preso ad esempio un sistema a 64 bit, la memoria fisica non usa esclusivamente tutti e 64 bit ma la memoria fisica sul sistema è molto inferiore rispetto a l'indirizzamento logico possibile con 64 bit! → alla quale corrisponde una tabella delle pagine che sarà quindi di dimensione più limitata! (con il vantaggio che tutti i processi usano la stessa tabella!)



→ cosa sarà memorizzato nella page table?

Viene indicato da quale processo è usato il frame → il processo è indicato con il pid al quale è associata una particolare pagina → per cui quando viene generato un indirizzo logico dobbiamo prima ricavare quale sia il pid del processo in esecuzione, poi verrà usata la pagina richiesta → tramite la coppia pid-pagina viene fatta una ricerca per cercare questa associazione e sulla base della posizione su cui viene trovata la pagina → si ricava dove è il frame! (associato) → al quale come al solito viene aggiunto l'offset per generare l'indirizzo fisico! → cosa importante da ricordare è che ad ogni riga delle tabelle delle pagine è associato un frame! → ad esempio il frame[2] nella page table indica quale processo lo usa e a quale pagina è indirizzato

+Dove sarà salvato il pid? Boh... forse in un particolare registro nel PCB → tramite la coppia pid-pagina viene quindi fatta una ricerca lineare per cercare a quale frame questa associazione corrisponde! → trovata la riga ne troviamo il frame! (come al solito se è trovata la corrispondenza pid-pagina cercata si lancierà un eccezione...)

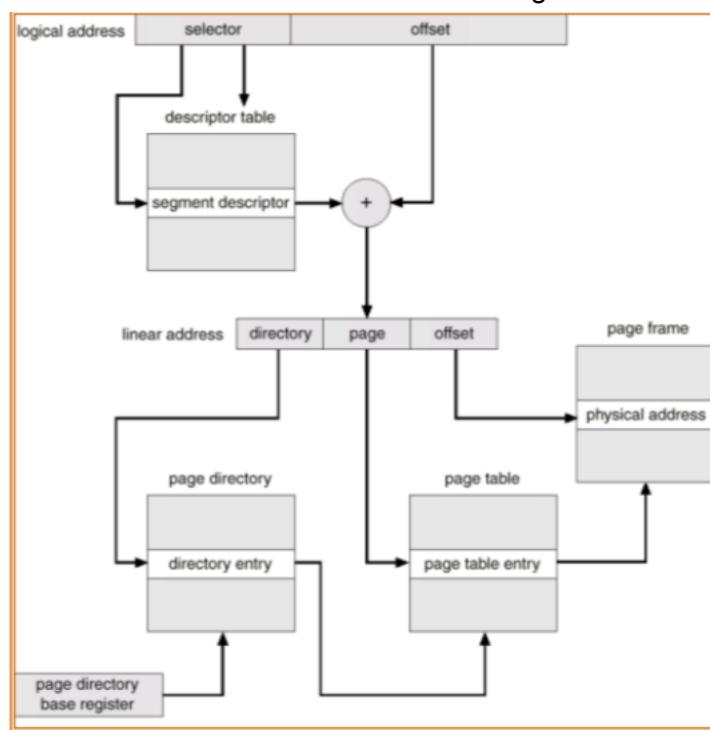
+Può essere usata una tabella hash per limitare la ricerca a una o poche righe della tabella → allo stesso modo l'uso della TLB diventa fondamentale per evitare tutti questi accessi alla tabella delle pagine e dunque migliorare le prestazioni

+Un problema però di questa gestione è il fatto che risulta difficile condividere due a pagine a processi diversi! → dovrei infatti mettere nella stessa riga 2 associazioni alla stessa pagina → potrei fare ad esempio una lista, ma complicherebbe le cose...

+Anche per la TLB non va tutto a rose e fiori... quando infatti viene fatto il context switch si deve cambiare l'informazione relativa alla gestione della memoria → cambiando però la tabella delle pagine il contenuto (associato ad una particolare riga) della TLB non risulta più valido, in quanto è presente solo pagina/frame e non anche il processo! → per cui quando cambio processo il processore dovrà buttare via il contenuto della TLB → e questo comporta che nei primi accessi accessi in memoria dovrà farli direttamente alla tabella delle pagine e poi portare il contenuto alla TLB → non può infatti sfruttare un eventuale uso pregresso!  
→ problema che però non diventa evidente nel caso dei thread all'interno di uno stesso processo! → infatti in quel caso lì non risulta necessario aggiornare la TLB poiché i thread condividono la stessa memoria → senza generare problemi di miss ecc... (per cui nel caso di context switch → se ci sono altri thread conviene fare prima quelli??)

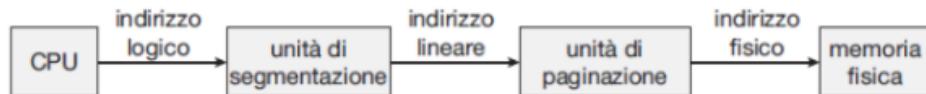
## Segmentazione paginata

Per via della grande numerosità dei codici scritti in architettura x86, i quali usavano la segmentazione come gestione della memoria, per risolvere il problema della frammentazione esterna si è pensato di fondere la segmentazione alla paginazione!  
→ Ovvero i vari processori vicini ad oggi continuano ad avere l'unità di segmentazione, soltanto che successivamente l'indirizzo generato viene poi paginato!



→ si parte dall'indirizzo logico formato da selector - offset → con il quale viene generato un indirizzo intermedio → il quale verrà alla fine paginato → il sistema operativo dunque gestisce l'allocazione della memoria effettiva usando l'allocazione, quindi ha libertà di posizionare tutti i frame usati in tutte le pagine del processo come vuole in memoria! → però poi il processore genererà indirizzi in segmenti! → per cui risulta compatibile con l'approccio segmentato!

→ l'architettura intel IA-32 risulta avere questa doppia modalità di gestione della memoria!



+ Per la **segmentazione** abbiamo quindi:

- 16K segmenti → numero di segmenti e non le dimensioni → di cui 8K sono detti locali del processo (Local Descriptor Table - LDT) → mentre i restanti 8K sono divisi fra tutti i processi tramite un'unica tabella detta (Global Descriptor Table) → per cui in sostanza ho una tabella dei segmenti per ogni processo, più una tabella per tutti i processi!
- Ciascun elemento della LDT e GDT usa 8 byte (con all'interno indirizzo base, limite ecc...)
- Indirizzo logico che composto da un selettore di segmento che usa 16bit → divisi a loro volta in s = 13 bit (id segmento), g = 1 bit (per indicare se è local o global) e infine p = 2 bit (per indicare il privilegio di accesso); successivamente il resto dell'indirizzo logico è composto da l'offset nel segmento a 32 bit → 4GB di segmenti

+ Per la **paginazione** invece:

- Paginazione a due livelli:
  - 10 bit per **directory delle pagine**
  - 10 bit per **offset nella directory delle pagine**
  - 12 bit **offset nella pagina**
- + Risulta presente all'interno del processore un registro detto CR3 che indica indirizzo della directory delle pagine (quella iniziale → la root dell'albero)
- + Si possono avere pagine da 4KB o da 4MB indicato da flag PageSize nel elemento directory delle pagine (con 4MB offset è di 22 bit) → in particolare sono fusi i 10 e i 12 bit per indicare l'offset
- Abbiamo in questo modo 4GB di memoria massima usabile! → ad un certo punto non abbastanza per gestire tanti processi e dunque tanta memoria → viene introdotta la PAE → Page Address Extension, in cui viene fatto una paginazione a 3 livelli (2 bit, 9 bit, 9bit, 12 bit)
- + Viene poi cambiata tabella delle pagine con id frame a 24 bit invece di 20 bit (attenzione non cambia la grandezza dei frame ma il loro numero!) → 24+12 = 36 bit di indirizzo(fisico) → max 64GB (ma ogni processo max 4GB per ogni segmento) → non è abbastanza!

## Architettura x86 - 64

La versione utilizzata oggi è stata inizialmente prodotta da AMD (e non da intel) → ma poi adottata anche da intel!

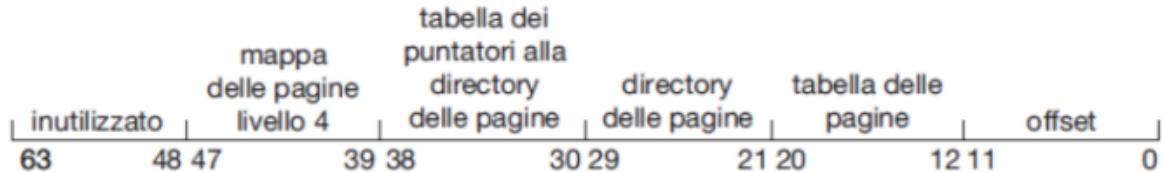
→ con 64bit sarebbero indirizzabili 16exabyte → per piefra un roba allucinante → per cui in realtà di questi 64 bit se ne usano solo 48! (al massimo 256 TB di memoria fisica)

→ si usa una paginazione gerarchica a 4 livelli costruiti come:

- 9 bit mappa delle pagine livello 4
- 9 bit tabella dei puntatori alla dir. delle pagine
- 9 bit directory delle pagine
- 9 bit tabella delle pagine

→ e alla fine 12 bit di offset! → ma sono presenti situazioni anche miste, in cui l'offset viene aggregato ad una certa parte della tabella delle pagine! → si possono quindi avere pagine da 4KB, 2MB (9+12) oppure 1GB (9+9+12)!

+Infine è possibile fare l'architettura con il PAE → tramite indirizzi fisici di 52 bit! → forse 256 TB non erano abbastanza... per cui si arriva a 4096 TB

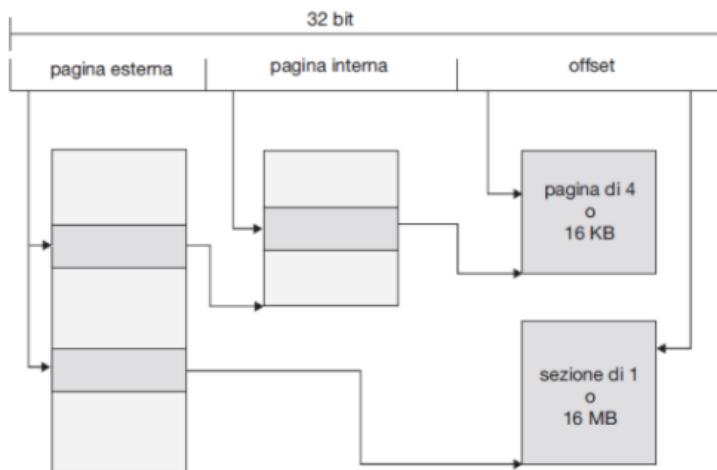


### Architettura ARM

Architettura usata nei sistemi mobili → iPhone, iPad, Android → anche qui abbiamo la gestione della memoria paginata con indirizzi fissi a 32 bit

→ Pagine da 4KB (12bit) o 16 KB (14bit) (paginazione a 2 livelli) oppure

→ Pagine da 1MB (20bit) o 16MB (24bit) → dette sezioni (paginazione a 1 livello)



→ ci sono situazioni in cui alcune parti usano come offset sia la pagina interna che l'offset stesso! → il vantaggio di ARM è il fatto di non dover essere compatibili con x86 per cui avevano maggiore libertà!

+Considerazione sul context switch già dette prima che copio e incollo:

- Come già detto durante il context switch tra diversi processi il sistema operativo deve impostare i registri della MMU del processore in modo che sia coerente con il processo da eseguire
- In particolare nel caso della paginazione deve anche invalidare la TLB altrimenti rischia di prendere l'associazione pagina→frame del processo precedente, ma invalidare la TLB porta che necessariamente si avranno molti «miss» e quindi un maggiore tempo di esecuzione
- Ma nel caso di processi multi-thread che condividono la memoria questi condividono anche la tabella delle pagine e quindi il context switch tra thread diversi di una stessa applicazione ha un costo minore (la TLB non viene invalidata) per questo quando il SO può scegliere in genere favorisce lo switch verso un thread della stessa applicazione.

# 21/05/2021

## Memoria Virtuale

Pensiamo di aver un processo che deve essere allocato... a priori gli sarà data tutta la memoria che al massimo lui potrà usare! → supponiamo quindi che ad esempio un processo usi al massimo 100 MBytes di memoria e ho 4GB di memoria a disposizione → di questi processi ne potrò mettere al massimo 40 processi

→ e se invece questi processi usano in ogni momento un dimensione di memoria più ridotta... ad esempio 10MB → infatti per eseguire un processo questo non ha bisogno di tutto lo spazio disponibile! → per cui con la stessa quantità di memoria disponibile potrei mettere in esecuzione ben 400 processi! → non considerando la necessità massima ma solo quella di picco che un processo ha bisogno per funzionare!

→ è infatti possibile che un processo abbia bisogno di tanta memoria solo in determinati momenti → ad esempio per eseguire una ricerca in una grande tabella... → finita questa fase può aver meno bisogno di risorse di memoria!

→ La Memoria virtuale si basa su questo fatto → ovvero di garantire comunque la possibilità di esecuzione in modo da soddisfare il loro bisogno di memoria (più ottimizzato rispetto al massimo)

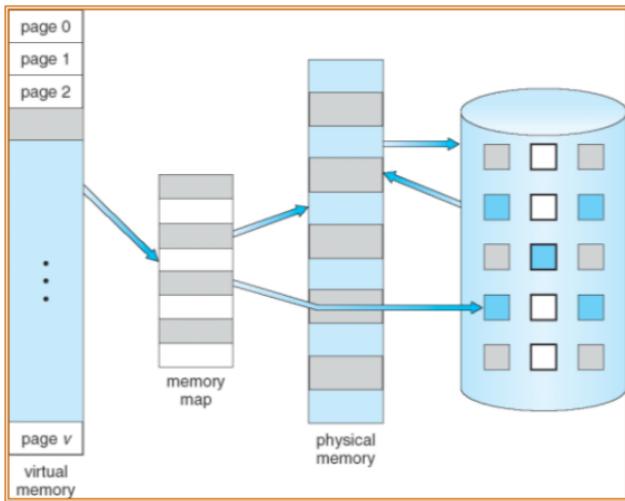
→ come visto precedentemente lo spazio di indirizzi logico può essere molto più grande dello spazio di indirizzi fisico effettivo (specialmente nel caso a 64 bit)

→ Dunque la memoria virtuale ha come valore principale quello di poter aumentare il suo grado di multiprogrammazione → di avere su un sistema più processi in esecuzione contemporaneamente

→ utile per esempio nei sistemi in cui devono essere eseguiti migliaia di transazioni al secondo... poter avere più processi in esecuzione sarà fondamentale!

+Come si implementa la memoria virtuale?

Si basa sulla paginazione! → detta in particolare paginazione su richiesta → dove la cosa principale che viene usata è il bit di validità visto in precedenza!



→ memoria virtuale vista come un'estensione della memoria fisica! → memorie virtuali che sono mappate appunto una mappa sulla memoria fisica → e quello che non può essere tenuto sulla memoria fisica in quanto limitata rispetto alle necessità totali → allora viene tenuto sul disco! → dispositivo di memorizzazione di massa il quale è MOLTO più lento rispetto alle altre memorie

### Paginazione su richiesta

Porta una pagina di memoria, solo quando risulta necessario! → dunque solo quando una pagina risulta necessaria la và a prendere dal disco e la tiene in memoria fino a che c'è disponibilità!

→ Per cui quando una pagina diventa necessaria si *referenzia* → processo che tenta di accederci → se il riferimento è sbagliato si ferma → segnale di *abort*!

→ se invece in quel momento non si trova in memoria (infatti il sistema operativo terrà traccia di cosa si trovi o meno in memoria) → allora si porta la pagina in memoria!

+Come capire se una pagina si trovi o meno in memoria?? → si usa il bit di validità!

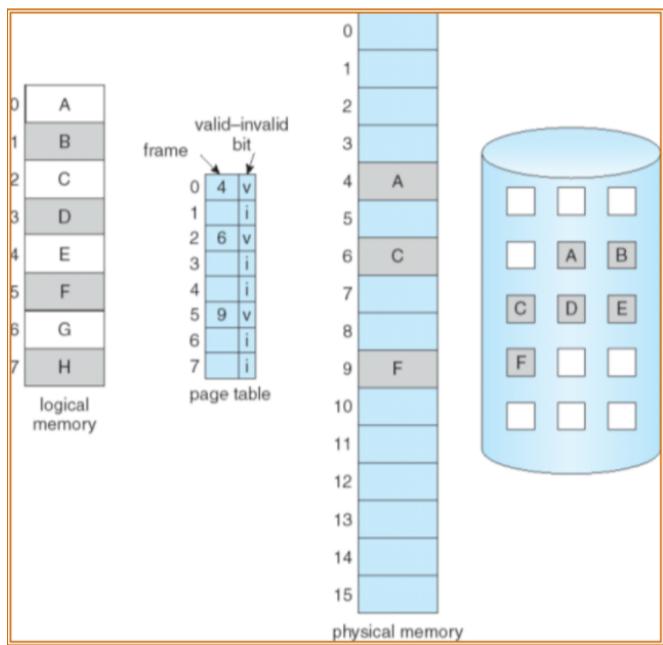
→ bit di validità presente nelle pagine che ci dice se quel frame lì è attualmente valido o meno! → se quindi la pagina a qui il processo fa riferimento risulta in memoria o no...

| Frame # | valid-invalid bit |
|---------|-------------------|
| 1       | 1                 |
| 1       | 1                 |
| 1       | 1                 |
| 1       | 1                 |
| 0       |                   |
| :       |                   |
| 0       |                   |
| 0       |                   |

page table

+Esempio:

Supponiamo di avere un processo con una certa memoria logica...



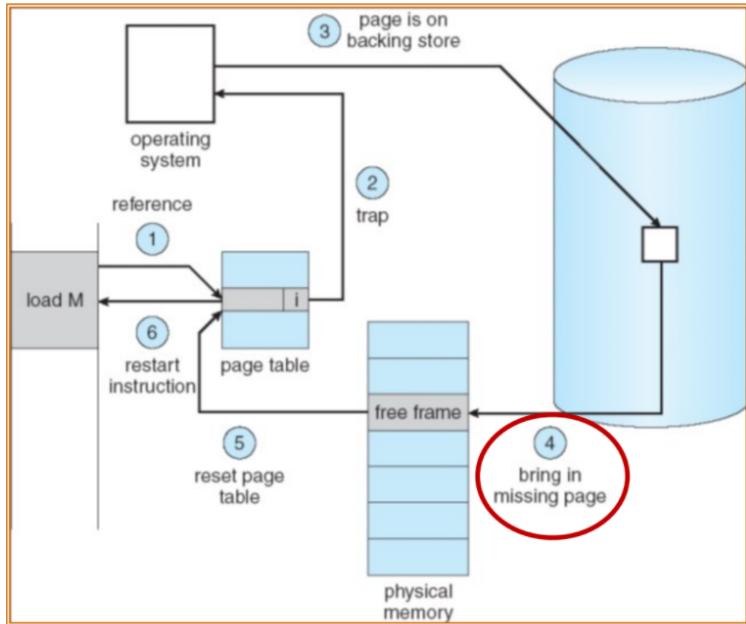
→ le pagine valide, sono la A, C e F → le altre non lo sono! → per cui se il processo accede alle pagine valido tutto ok → quando invece farà riferimento alla pagina B (o pagina 1) → va nella tabella delle pagine e trovando il bit invalido, allora capisce che non è stato associato alcun frame alla pagina → abbiamo il cosiddetto *page fault* che dovrà fare in modo di rendere valida quella posizione!

## Page fault

Se viene riferita una pagina il primo riferimento genera una trap nel sistema operativo → page fault → a quel punto il sistema operativo decide se il riferimento non è valido → in quanto comunque può esserci stato un errore nell'allocazione della memoria per un processo, dunque il riferimento è sbagliato → oppure se la pagina è valida allora la pagina non risulta in memoria in quel momento!

→ Per cui si decide di prendere un frame libero nella memoria fisica → a quel punto inizia il caricamento della pagina desiderata nel frame → “si va nel disco e si fà la richiesta, dammi il contenuto che devo mettere in questa pagina” → per cui ciò che ottiene lo mette nel frame libero → a quel punto imposta il bit di validità di quell'entry di quella tabella delle pagine a 1 → infine si fà ripartire l'istruzione fallita !

→ per cui quando accade un page fault, non si perde l'istruzione che si è cercato di eseguire ma il gestore delle interruzioni che gestisce questa trap → entra in esecuzione per gestire questa cosa! → Riassumendo:



→ nel passaggio da 2 a 3 come si diceva prima... il sistema operativo che l'indirizzo logico prodotto dalla CPU sia un indirizzo valido → e dunque da poter ricercare in memoria!  
 → nel passaggio 5 invece, nella page table verrà mappata la nuova riga con il numero del frame che era libero (riferimento al frame) e a quel punto verrà fatto ripartire l'istruzione! → che stavolta farà riferimento ad una pagina valida dunque riesce ad accedere in memoria!

+Tutto bello e fantastico se non però per il fatto di dover accedere nel disco! → il tempo che passa la richiesta della pagina da prendere dal disco al momento in cui il contenuto di questa pagina arriva effettivamente è molto alto! → dell'ordine dei millisecondi!  
 → tempo di gestione del page fault che è quindi dominato da questo passaggio qua! → un riferimento ad una pagina che non è valida impiega molto tempo per essere risolta → per cui in realtà la CPU non sta lì ad aspettare tutto il tempo ma se è già pronto un altro processo da eseguire passa a quello! → sperando che questi non debbano fare delle richieste a delle pagine non presenti in memoria... infatti nel caso tutti i processi non riescano a trovare le loro pagine in memoria e quindi tutti vanno sul disco → dunque non si può far altro che aspettare per molti millisecondi (invece che nano) → il sistema diventa quindi inutilizzabile! (questo succede soprattutto nel caso in cui vengono fatti agire molti processi in contemporanea ma la memoria è molto poca... il sistema diventa troppo lento!)

### Sostituzione di una pagina

+Un problema sarà quindi quello di gestire lo spazio di memoria libero → potremmo infatti arrivare ad un momento in cui tutti i frame della memoria fisica sono pieni → occupati dai processi che sono in esecuzione → quale scegliere quello da riusare... ne dovrò scegliere qualcuno da togliere! → come sceglierlo?  
 → Fra le diverse varianti presenti ne dovrò scegliere quella che genera meno page fault!

### Performance della paginazione su richiesta

Consideriamo la frequenza di Page Fault (p) → ovvero quanti page fault si hanno in un'unità di tempo → con p compreso fra 0 (nessun page fault) e 1.0 (ogni riferimento provoca un page fault)

→ Quale sarà il tempo effettivo di accesso?

EAT =  $(1 - p) \times$  accesso in memoria +  $p$  (tempo page fault + [tempo swap page out] + tempo swap page in + tempo restart istruzione)

→ il costo di accesso in memoria sarà quello normale e non quello di accesso al disco!

→ nel caso di page fault invece si avrà un tempo di page fault base (sia per la gestione dell'interruzione, che altre istruzioni...) → poi c'è un eventuale tempo di swap page out nel caso la pagina scelta come sacrificabile sia stata modificata da un certo processo → per cui in quel caso la dovrò pure salvare (sul disco)!

→ poi c'è sia un tempo per caricare dentro la pagina che per fare il restart dell'istruzione!

→ come al solito i tempi che dominano sono quello di *page out* e *page in* proprio perchè sono a colloquio con il disco!

+Esempio:

→ Tempo accesso in memoria = 200 nano secondi e Tempo di gestione page fault = 8 millisecondi → notare già qui la grande differenza...

Calcolo l'EAT → EAT =  $(1 - p) \times 200 + p \times (8 \text{ millisecondi}) = (1 - p) \times 200 + p \times 8.000.000 = 200 + 7.999.800 \times p$  (in nanosec)

→ per cui preso ad esempio → se 1 riferimento su 1000 genera page fault avrà un

→ EAT = 8,2 microsecondi, rallentato di 40 volte!

+Supponiamo invece di voler ricavare il page fault rate per avere un EAT ridotto almeno del 10% →

- EAT <  $200 \times (1 + 10\%)$
- $200 + 7.999.800 \times p < 220$
- $7.999.800 \times p < 20$
- $p < 0,00000025$
- 1 accesso ogni 399.990 deve generare page fault

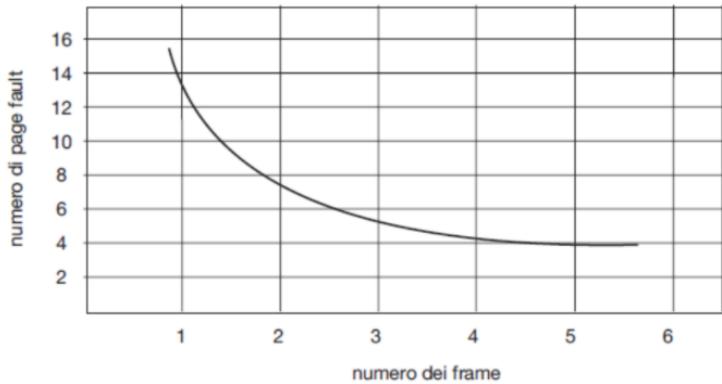
→ per cui dovrei avere che per ogni 400mila accessi 1 solo accesso può generare un page fault per avere una degradazione di performance accettabile!

+Torniamo alla sostituzione di una pagina → in generale il frame da riusare va prima riscritto sul disco e poi sostituito con il nuovo frame (soprattutto per non perdere le ultime modifiche possibili) → questo sarà possibile solo se la CPU dispone di un bit di modifica che ci dice che quella pagina lì è stata modificata dal processo...

→ se il processore non ha questa possibilità allora la cosa si fa difficile! → perchè non sai se quel processo che ha usato quella pagina c'ha scritto o meno sopra (processori di oggi infatti danno la possibilità di capire se la pagina è stata modificata o meno!)

+Consideriamo per ogni processo la sequenza delle pagine usata da un processo, non considerando i riferimenti immediatamente successivi alla stessa pagina → ovvero se un processo fa riferimento alla stessa pagina più volte, allora viene considerato come una singola pagina!

→ in generale aumentando il numero di frame il numero dei page fault andrà a diminuire!



(giustamente dice pierfra quattro

frame e basta non esistono...)

## Sostituzione FIFO

Come si comporteranno i vari metodi di sostituzione?? → incominciamo dal FIFO → dove si decide di sostituire la pagina presente da più tempo!

Per esempio:

Es. successione riferimenti:

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Memoria con 3 frame

- 7 → 7 xx
  - 0 → 0 2 3
  - 0 → 7 0 x
  - 3 →
  - 1 → 7 0 1
  - 2 → 2 0 1
  - 1 → 0 1 3
  - 0 →
  - 2 → 0 1 2
  - 3 → 2 3 1
  - 0 →
  - 1 →
  - 4 → 4 3 0
  - 7 → 7 1 2
  - 2 → 4 2 0
  - 0 → 7 0 2
  - 3 → 4 2 3
  - 1 → 7 0 1
- 15 page faults**

→ riferimenti multipli alla stessa pagina (ad esempio una sequenza di istruzioni all'interno di un solo codice) si considerano come un solo riferimento → ad esempio più istruzioni di registro sulla pagina 7 → avrò sempre riferimenti alla stessa pagina che non generano page fault! → i valori ripetuti vengono collassati in un unico riferimento!

→ le frecciette rosse sono tutti page fault! → alla richiesta della pagina 2 → esauriti i frame il primo ad uscire sarà il primo entrato → ovvero la pagina 7! e così via per le altre richieste non presenti in memoria!

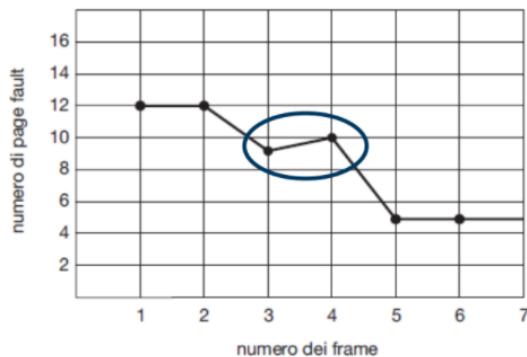
→ il numero di page fault è ovviamente elevato dovuto al numero molto piccolo dei frame...

## Anomalia di Belady

Può accadere invece che con alcune sequenze di riferimenti aumentando il numero di frame in realtà aumentano il numero di page fault! (ovviamente non accade con tutte le sequenze)  
Ad esempio → presa la sequenza 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5; con:

- 3 frame → 9 page faults  
→ 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 frame → 10 page faults  
→ 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

→ il diagramma dice infatti → l'adndamento aumenta!



## Sostituzione ottimale

Si sostituisce la pagina che non verrà trovata per il più lungo periodo di tempo → si guardà al futuro ! → dunque si cerca di rendere minimo il tasso di page fault perchè si va a cercare fra le pagine che ci sono in memoria quale sarà l'ultima ad essere usata rispetto alle altre → per cui si toglie quella! → anche se effettivamente non lo puoi stabilire con certezza...

→ dunque principalmente il suo utilizzo è al fine della valutazione → ovvero quanto gli altri algoritmi si avvicinano alla soluzione ottimale!

+Ad esempio con la stessa sequenza di prima cosa accade?

Es. successione riferimenti:

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Memoria con 3 frame

- 7 → 7 xx
  - 0 → 7 0 x
  - 1 → 7 0 1
  - 2 → 2 0 1
  - 3 → 2 0 3
  - 0 →
  - 4 → 2 4 3
  - 2 →
  - 3 →
  - 0 →
  - 1 →
  - 7 → 7 0 1
  - 0 →
  - 2 →
  - 1 →
- 9 page faults**

→ i primi 3 riferimenti ho 3 frame liberi quindi non ho problemi, al quarto invece fra 7,0 e 1 quale quello a cui farò riferimento il più lontano possibile? → guardo la successione e trovo che è la pagina 7! → sacrifico quella! → stesso discorsi per gli altri casi di page fault!

→ si passa da 15 a "solo" 9 page fault!

+Non è possibile sapere la sequenza dei riferimenti a priori proprio perchè questi dipendono dalle istruzioni e i dati che verranno man mano letti... solo dopo aver ricreato la successione dei riferimenti allora ne puoi fare una valutazione!

## Sostituzione LRU

Stavolta tra tutte quelle che abbiamo in memoria decidiamo di sostituire quella che è stata usata meno di recente (Least Recently Used) → poichè appunto significa che è tanto che non ci si accede, dunque si ipotizza che non ci si faccia più riferimento nel futuro!

+Riprendiamo sempre la stessa sequenza:

- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Memoria con 3 frame

|             |             |                                   |
|-------------|-------------|-----------------------------------|
| ▪ 7 → 7 xx  | ▪ 4 → 4 0 3 | ▪ 2 →                             |
| ▪ 0 → 7 0 x | ▪ 2 → 4 0 2 | ▪ 0 → 1 0 2                       |
| ▪ 1 → 7 0 1 | ▪ 3 → 4 3 2 | ▪ 1 →                             |
| ▪ 2 → 2 0 1 | ▪ 0 → 0 3 2 | ▪ 7 → 1 0 7 <b>12 page faults</b> |
| ▪ 0 →       | ▪ 3 →       | ▪ 0 →                             |
| ▪ 3 → 2 0 3 | ▪ 2 →       | ▪ 1 →                             |
| ▪ 0 →       | ▪ 1 → 1 3 2 |                                   |

→ quando vogliamo far entrare la pagina 3, la sostituisco con la 1, poichè la pagina 0 anche se entrata dopo è stata richiamata nel riferimento precedente! → e così via per le altre...

notare che possono accadere casi in cui ad esempio decido di sostituire la pagina 2 e subito dopo è richiesto un richiesto un riferimento alla pagina 2! → page fault!

→ in totale abbiamo 12 page fault → dunque meglio rispetto a fifo, ma non bene quanto quella ottimale!

+In particolare sostituzione ottimale e LRU non soffrono della *anomalia di Belady* → aumentando il numero di frame il numero di page fault non peggiora!

+Sostituzione LRU dunque risulta quella più usata → basandosi però sui riferimenti non è molto facile da applicare → in quanto per uso di una pagina si intende sia di lettura che di scrittura di una pagina → e quindi ne dobbiamo tenere traccia! → come fare questo?

Possiamo ad esempio usare un **contatore** → il quale verrà incrementato ad ogni accesso in memoria → il valore di questo contatore verrà quindi associato ad ogni uso di una pagina!

→ uso quindi una pagina? salvo sia il contenuto di questa che il valore del contatore!

→ quando dovrò scegliere quale pagina rimpiazzare sceglierò la pagina con il valore più piccolo → in quanto sarà la pagina che è stata usata meno di recente!

+L'altra possibilità invece è tramite l'uso dello **stack** → ogni volta che una pagina viene usata viene inserita in testa allo stack (oppure spostata se già presente nello stack) → per cui infondo ci sarà quella che sarà stata meno usata di recente!

→ l'aggiornamento dello stack vā fatto ad ogni riferimento in memoria! → e per aggiornare lo stack ci vorrà una serie di istruzioni... non proprio semplice!

→ conviene dunque usare l'implementazione hardware con il contatore! → anche questa leggermente costosa...

→ per evitare questi problemi si fanno le *pseudo LRU* → le quali non trovano proprio il riferimento più lontano nel tempo ma una loro approssimazione! → si basano su di un bit che indica se la pagina è stata "ripetuta" → sulla base quindi dei bit di utilizzazione il sistema operativo può andare a guardare quali pagine sono state utilizzate in un certo lasso di tempo! → dunque ad esempio ogni 10-100 millisecondi (in modo regolare) controllare quali risorse sono state utilizzate e quindi resettare questo valore! → se quindi in quel lasso di tempo il bit è stato settato a 1, si incrementa il contatore! → per dire quante volte una pagina è stata usata! → e tramite il contatore si stabilisce quale è stata usata meno di recente...

→ questa è una soluzione intermedia poiché non vado a guardare i valori dei contatori ad ogni riferimento in memoria, ma ogni tanto vado a vedere se le pagine sono state usate! (in particolare se riconosco che le pagine sono usate, incremento il contatore ma resetto il bit)

→ come al solito qualcosa di impegnativo dal punto di vista dell'hardware... anche se in

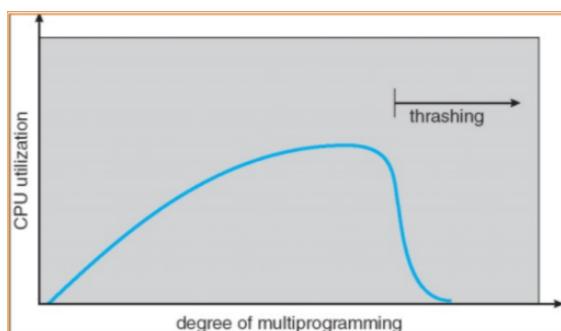
questo modo si ha un'idea dell'uso delle pagine da parte dei processi

### Trashing

Se un processo non ha abbastanza pagine in memoria, la frequenza dei page fault è molto alta! → ciò implica che il sistema operativo sarà impegnato principalmente nel portare le pagine dal disco alla memoria... ma in questa fase la CPU sarà in una fase di bassa utilizzazione, in quanto sta aspettando → le varie istruzioni in esecuzione sono tutte in attesa della pagina dal disco!

→ il sistema operativo dunque vedendo la CPU libera, decide di mettere altri processi in esecuzione aumentando quindi il grado di multiprogrammazione! → ma aggiungendo un altro processo diminuisce ancor di più la memoria a disposizione → che porterà ad un ulteriore peggioramento delle performance del sistema

→ si parla quindi di *trashing* nel caso il sistema operativo sia occupato quasi esclusivamente a fare swap delle pagine da/a disco!



→ dove si aumenta il numero dei processi in esecuzione → dunque aumento l'uso della CPU → fino ad un punto in cui diminuisce improvvisamente poiché non c'è abbastanza memoria per tenere tutti i processi e quindi questi cominciano ad usare meno CPU in quanto sono in attesa della pagina sul disco! → quindi l'uso della CPU diminuisce fino a raggiungere lo zero se tutti i processi hanno avuto un page fault!

→ dunque è possibile avere una situazione in cui ho un sacco di processi in esecuzione ma la CPU risulta inutilizzata poiché non c'è abbastanza memoria!

+Come risolvere questo problema?

→ Se un processo entra in una situazione in cui genera spesso page fault → ovvero quella del trashing → allora il sistema operativo gli può dire "tu stai generando troppi page fault → dunque usi troppa memoria... allora ti limito! → te puoi riusare solo i tuoi frame → quelli che hai te li tieni e se vuoi riusare qualcosa casomai riusa i tuoi!"

→ gli altri processi dunque hanno più possibilità in quanto la parte restante dei frame a disposizione possono essere utilizzati! → dunque è più difficile che il resto abbia problemi di trashing (infatti può bastare anche solo un processo per avere trashing → gli altri non trovano disponibile la memoria perché gli viene tolta e quindi iniziano anche loro a generare page fault!)

+Un altro modo di risolvere il problema:

→ Il trashing si evita se ogni processo in esecuzione ha in memoria i frame che gli servono in quel momento (**working-set**) generando pochi page fault → se un processo ha in memoria il suo working set questo genera pochi page fault!

→ questo ovviamente implica avere una memoria abbastanza capiente in grado di tenere il working-set dei vari processi → se però il suo working-set è comunque incredibilmente grande rispetto alla memoria disponibile (ad indicare che il processo ha bisogno di molta memoria) → allora si provocano necessariamente molti page fault!

## Working-set

→ working set definito come insieme delle pagine riferite dal processo in un certo numero di riferimenti in memoria → per cui la dimensione del working set cambia durante l'esecuzione del processo → in particolare si dirà che il processo si sposta di "località" → in un certo momento usa il working di una certa dimensione, poi può cambiare poiché si ha un'altra fase e dunque aumentano le dimensioni  
→ il sisOp dovrà dunque tenere traccia delle dimensioni di ogni working set, dei processi che sono in esecuzione! → in particolare il sistema non sarà in trashing:

- la somma delle dimensioni dei working set è inferiore al numero di frame disponibili  
→ ciò indica che si riesce a tenere in memoria il working set di tutti i processi che sono in quel momento in esecuzione! → tutti i frame dei processi stanno in memoria  
→ avrà pochi page fault → dunque non ho trashing!

→ se però ad un certo momento il sisOp nota che un certo processo sta facendo riferimento a più zone di memoria, dunque aumentando il suo working set e sommando questo con quello di tutti gli altri vede si supera il numero dei frame disponibili → allora il sisOp decide di **sospendere** quel processo problematico (o quei processi) → viene liberato TUTTO lo spazio usato dal processo prima in modo da essere utile agli altri fino al momento in cui diventa disponibile la richiesta di memoria del processo sospeso e lo si fa ripartire!  
+Altra politica sarà quello di farlo terminare direttamente... (cosa che avviene nei sistemi linux)

+Htop e gestione attività... si può controllare quale sia il working set di ogni processo → e in particolare si nota che google ha diversi working set per ogni scheda aperta! → abbiamo un processo associato ad ogni tab... in modo che se uno non funziona non muore l'intera "applicazione" ma solo quel tab → maggiore stabilità!

+Memoria virtuale come illusione a tutti i processi di avere un sacco di memoria che non in realtà non c'è! → tramite il disco si usa per tenere quella parte di memoria che non può essere tenuta nella memoria centrale!

FINE → il resto sono esercitazioni per l'esame...