

SISTEMI OPERATIVI - STAIANO

UNIVERSITÀ DEGLI STUDI DI NAPOLI
“PARTHENOPE”



Appunti a cura di
FIORENTINO MICHELE

SISTEMI OPERATIVI – PARTE 1

5. LEZIONE 1/2 – COS'È UN SISTEMA OPERATIVO

- 6. OBIETTIVI DI UN SO
- 7. FUNZIONI DI UN SO

10. LEZIONE 3 (1) – PANORAMICA SUI SO

- 10. AMBIENTI DI CALCOLO E NATURA DELLE ELABORAZIONI
- 12. EFFICIENZA, SERVIZIO PER L'UTENTE e PRESTAZIONI DEL SISTEMA
- 13. CLASSI DI SISTEMI OPERATIVI
 - 13. ELABORAZIONE BATCH
 - 14. MULTIPROGRAMMATI
 - 16. SISTEMI TIME-SHARING
 - 17. SISTEMI OPERATIVI REAL-TIME
 - 19. SISTEMI OPERATIVI DISTRIBUITI

21. LEZIONE 3 (2) – STRUTTURA DEI SO

- 21. FUNZIONAMENTO DI UN SO
- 22. POLITICHE E MECCANISMI, PORTABILITÀ ed ESPANDIBILITÀ DEI SO
- 23. SO CON STRUTTURA MONOLITICA / A STRATI
- 24. SO BASATI SU KERNEL

26. LEZIONE 4 – PROCESSI

- 27. PROCESSI E FIGLI
- 28. CONCORRENZA E PARALLELISMO
- 29. STATO DI UN PROCESSO e TRANSAZIONI DI STATO
- 31. CONTESTO DI UN PROCESSO e PROCESS CONTROL BLOCK
- 32. SALVATAGGIO DEL CONTESTO, SCHEDULING e DISPATCHING
- 33. GESTIONE DEGLI EVENTI
- 35. CONDIVISIONE, COMUNICAZIONE e SINCRONIZZAZIONE TRA PROCESSI

36. LEZIONE 5 – THREAD

- 37. TRANSIZIONI DI STATO
- 38. VANTAGGI THREAD vs PROCESSI
- 39. CODIFICA PER USARE I THREAD
- 39. THREAD DI LIVELLO KERNEL, UTENTE e IBRIDI

42. LEZIONE 6 – SINCRONIZZAZIONE DEI PROCESSI (1)

- 43. RACE CONDITION
- 44. SEZIONI CRITICHE
- 46. SINCRONIZZAZIONE DI CONTROLLO e OPERAZIONI INDIVISIBILI
- 48. APPROCCI ALLA SINCRONIZZAZIONE
- 49. STRUTTURA DEI SISTEMI CONCORRENTI

51. LEZIONE 7 – SINCRONIZZAZIONE DEI PROCESSI (2)

- 52. Produttori-Consumatori con buffer limitati
- 54. Lettori e Scrittori
- 55. Filosofi a cena
- 56. APPROCCI ALGORITMICI PER LE SEZIONI CRITICHE
 - 57. ALGORITMI A DUE PROCESSI
 - 58. DECKER
 - 59. PETERSON
- 60. ALGORITMI A N PROCESSI

62. LEZIONE 8 – SINCRONIZZAZIONE DEI PROCESSI (3)

- 62. SEMAFORI
- 64. USO DEI SEMAFORI NEI SISTEMI CONCORRENTI
- 66. SOLUZIONI AI PROBLEMI DI SINCRONIZZAZIONE CON SEMAFORI
 - 67. Produttori-Consumatori con buffer limitati
 - 68. Lettori e Scrittori
- 71. IMPLEMENTAZIONE DEI SEMAFORI

72. LEZIONE 9 – SINCRONIZZAZIONE DEI PROCESSI (4)

- 72. I MONITOR
- 73. REGOLE PER LE VARIABILI DI CONDIZIONE
- 73. SEMAFORO BINARIO implementato con MONITOR
- 74. PRODUTTORI-CONSUMATORI CON MONITOR
- 75. PROBLEMA DEL BARBIERE ADDORMENTATO (classico prob. di sinc)
- 77. CASI DI STUDIO DI SINCRONIZZAZIONE DI PROCESSI

78. LEZIONE 11 – MESSAGE PASSING

- 78. COMBINAZIONI DI SEND e RECEIVE
- 79. INDIRIZZAMENTO e MAILBOX
- 80. ASSOCIAZIONE E PROPRIETÀ DELLE MAILBOX
- 80. FORMATO DEI MESSAGGI
- 81. MUTUA ESCLUSIONE con SCAMBIO DI MESSAGGI
- 82. PRODUTTORE-CONSUMATORE con SCAMBIO MESSAGGI
- 83. STRATEGIE DI ACCODAMENTO
- 83. SCAMBIO DEI MESSAGGI NEI VARI SO

84. LEZIONE 12 – SCHEDULING (1)

- 86. TERMINOLOGIA E CONCETTI DI SCHEDULING
- 87. TECNICHE DI SCHEDULING, RUOLO DELLE PRIORITÀ
- 88. SCHEDULING SENZA PRELAZIONE
- 90. SCHEDULING CON PRELAZIONE

93. LEZIONE 13 – SCHEDULING (2)

- 95. STRUTTURE DATI e MECCANISMI DI SCHEDULING
- 96. SCHEDULING BASATO SU PRIORITÀ
- 97. SCHEDULING ROUND-ROBIN CON TIME SLICING
- 97. SCHEDULING MULTILIVELLO
- 98. SCHEDULING REAL-TIME
- 100. SCHEDULING CON DEADLINE
- 102. SCHEDULING PER PROCESSI PERIODICI

104. LEZIONE 15 – DEADLOCK (1)

- 104. DEADLOCK NELL'ALLOCAZIONE DI RISORSE
 - 106. Grafi RRAG e WFG
 - 108. Matrice
- 108. GESTIONE DEI DEADLOCK
- 109. RILEVAMENTO E RISOLUZIONE DEI DEADLOCK
- 111. PREVENZIONE DEADLOCK

114. LEZIONE 16 – DEADLOCK (2)

- 114. EVITARE I DEADLOCK
 - 114. Algoritmo del banchiere
 - 119. Algoritmo del Banchiere con allocazioni parametriche

121. LEZIONE 17 – GESTIONE DELLA MEMORIA (1)

- 121. GERARCHIA DELLA MEMORIA
- 123. ALLOCAZIONE STATICHE E DINAMICA DELLA MEMORIA
- 124. ESECUZIONE DEI PROGRAMMI
- 127. ALLOCAZIONE DI MEMORIA AD UN PROCESSO: STACK ed HEAP
- 129. GESTIONE DELLO HEAP

133. LEZIONE 18 – GESTIONE DELLA MEMORIA (2)

- 133. ALLOCAZIONE CONTIGUA DELLA MEMORIA
- 133. ALLOCAZIONE NON CONTIGUA DELLA MEMORIA
- 135. ALLOCAZIONE CONTIGUA VS NON CONTIGUA
- 135. PROTEZIONE DELLA MEMORIA
- 136. PAGINAZIONE
- 139. SEGMENTAZIONE
- 140. SEGMENTAZIONE CON PAGINAZIONE

141. LEZIONE 20 – MEMORIA VIRTUALE (1)

- 142. MEMORIA VIRTUALE con PAGINAZIONE
- 143. Paginazione su richiesta: FAULT DI PAGINA
- 146. Paginazione su richiesta: SOSTITUZIONE DELLE PAGINE
- 147. ALLOCAZIONE DELLA MEMORIA AD UN PROCESSO
- 148. HARDWARE DI PAGINAZIONE
- 152. ORGANIZZAZIONE DELLE TABELLE DELLE PAGINE IN PRATICA

153. LEZIONE 22 – MEMORIA VIRTUALE (2)

153. POLITICHE DI SOSTITUZIONE DELLE PAGINE

 154. SOSTITUZIONE DELLE PAGINE OTTIMALE

 154. SOSTITUZIONE FIFO

 154. SOSTITUZIONE LRU

156. PROPRIETÀ DELLO STACK

157. PROBLEMI CON POLITICA FIFO

157. POLITICHE DI SOSTITUZIONE PAGINA IN PRATICA

158. ALGORITMI DI CLOCK

160. CONTROLLARE L'ALLOCAZIONE DI MEMORIA AD UN PROCESSO

160. WORKING SET

162. COPY-ON-WRITE

163. LEZIONE 23 – FILE SYSTEM

163. INTRODUZIONE FILE SYSTEM e IOCS

165. FILE

169. DIRECTORY

170. ALBERI DELLE DIRECTORY

170. GRAFI DELLE DIRECTORY

171. PROTEZIONE DEI FILE

171. ALLOCAZIONE DI SPAZIO SU DISCO

172. ALLOCAZIONE CONCATENATA

173. ALLOCAZIONE INDICIZZATA

174. INTERFACCIA TRA FS E IOCS, AFFIDABILITÀ e PERDITA DI CONSISTENZA DEI FILE SYSTEM

176. LEZIONE 24 – IOCS: INPUT OUTPUT CONTROL SYSTEM

177. ORGANIZZAZIONE DELL'I/O

181. DISCO MAGNETICO

182. ORGANIZZAZIONE DEI DATI SU DISCO

183. DISCHI RAID (Redundant Array of Independent Disks)

185. DRIVER DI DISPOSITIVO

186. SCHEDULING DEL DISCO

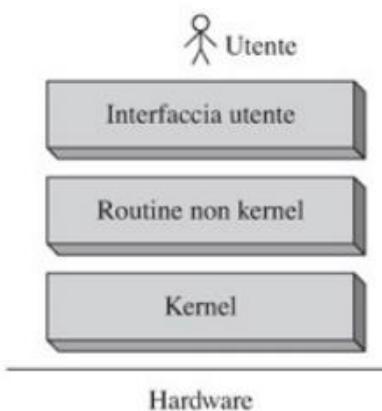
LEZIONE 1/2 – COS’È UN SISTEMA OPERATIVO

La definizione di Sistema Operativo varia in base a chi sta utilizzando il sistema di elaborazione, quindi dipende dalle necessità dell’utente.

Per alcuni utenti, l’utilizzo di un computer consiste semplicemente nella necessità di navigare in Internet o mandare e-mail, mentre per alcuni altri significa eseguire programmi per elaborare dati o eseguire elaborazioni scientifiche. Un sistema operativo deve soddisfare i bisogni di tutti i suoi utenti, pertanto deve fornire diverse funzionalità.

In *linea generale*, un **sistema operativo** è un software di base che gestisce le risorse hardware e software della macchina, fornendo servizi di base ai software applicativi.

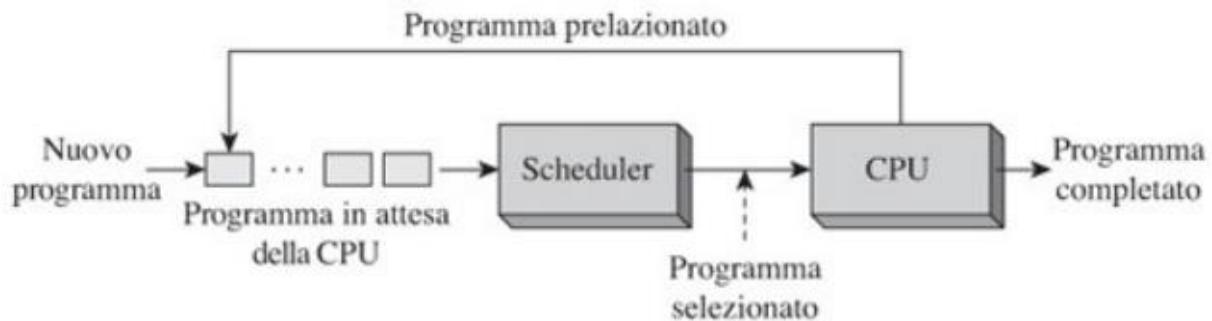
Dal **punto di vista di un progettista**, il sistema operativo è diviso in diversi strati, quali: *Interfaccia utente*, *routine non kernel*, e *kernel*.



- *Interfaccia utente*: l’interfaccia utente accetta comandi per eseguire programmi e usa le risorse e i servizi forniti dal sistema operativo. Può essere un’interfaccia a linea di comando (comandi utente attraverso prompt dei comandi) o un’interfaccia grafica (GUI, comandi utente attraverso click del mouse).
- *Routine non di sistema*: queste routine implementano i comandi utente relativi all’esecuzione dei programmi e all’uso delle risorse del computer; sono richiamate dall’interfaccia utente.
- *Kernel*: il kernel è il cuore del SO. Esso controlla il computer e fornisce un insieme di funzioni e servizi per utilizzare la CPU, la memoria e le altre risorse del computer. Le funzioni e servizi del kernel sono richiamati dalle routine non-kernel e dai programmi utente.

NB: la differenza fondamentale fra un sistema operativo e il kernel è che il sistema operativo agisce come un’interfaccia fra l’utente e il computer, mentre il kernel agisce come un’interfaccia fra il software e l’hardware del sistema.

Schema di scheduling



I programmi vengono inviati allo scheduler (organizzatore), il quale stabisce l'ordine attraverso cui verranno inviati i programmi alla CPU.

La CPU può quindi completare l'esecuzione del programma o può prelazionarlo.

Con *prelazione* si intende l'operazione in cui un processo viene temporaneamente interrotto e portato al di fuori della CPU al fine di permettere l'esecuzione di un altro processo.

OBIETTIVI DI UN SO

Gli obiettivi fondamentali di un SO sono: *uso efficiente, convenienza per l'utente e assenza di interferenze*.

Da notare come queste caratteristiche possano entrare in conflitto fra di loro. Ad esempio, l'enfasi sui servizi veloci potrebbe significare che le risorse come la memoria devono rimanere allocate anche quando il programma non è in esecuzione, portando ad un uso inefficiente delle risorse.

La soluzione consiste nel trovare un giusto compromesso fra la convenienza per l'utente e l'uso efficiente.

Uso efficiente

Un Sistema Operativo assicura un uso efficiente di memoria, CPU, e dispositivi di I/O. Si può avere poca efficienza se un programma non usa una risorsa che gli è stata allocata.

Da notare come il SO stesso consumi risorse di CPU e memoria, che costituiscono **overhead**, e che riduce le risorse per i programmi utente.

L'efficienza può essere assicurata attraverso il monitoraggio delle risorse da parte del SO, che aumenta tuttavia l'overhead, o attraverso l'applicazione di particolari strategie sub ottime.

Convenienza per l'utente

La “convenienza per l’utente” è costituita da molti aspetti, che sono andati ad evolversi nel corso del tempo.

Ad esempio, all’inizio era considerato “conveniente per l’utente” semplicemente eseguire un programma scritto in un linguaggio ad alto livello (**soddisfacimento delle necessità**), ma l’esperienza con i primi sistemi operativi ha portato alla richiesta di **servizi sempre migliori**, ovvero che fossero in grado di rispondere alle necessità dell’utente con velocità sempre maggiori.

Altri aspetti consistono nell’aver **interfacce** sempre più semplici da utilizzare, **nuovi modelli di programmazione**, come è stata la programmazione concorrente, **caratteristiche orientate al web** e l’aggiunta di caratteristiche e tecnologie sempre più nuove (**evoluzione**).

Assenza di interferenze

Con “assenza di interferenza” intendiamo le azioni prese dal SO per prevenire la corruzione delle attività computazionali eseguite dall’utente o dal SO stesso da parte di altre persone.

Tale prevenzione avviene allocando le risorse per un uso esclusivo di programmi e servizi del SO e prevendo l’accesso illegale delle risorse.

FUNZIONI DI UN SO

Le principali funzioni di un SO sono: *gestione dei programmi, gestione delle risorse, e sicurezza e protezione*.

Inoltre il SO esegue molte altre importanti operazioni, ad esempio costruisce una lista di risorse durante la fase di boot, memorizza informazioni per la sicurezza, verifica l’identità di un utente al momento del login, inizializza l’esecuzione dei programmi quando richiesto dall’utente, esegue l’allocazione delle risorse e ne preserva lo stato, preserva lo stato corrente dei programmi ed effettua lo scheduling.

Gestione dei programmi

Il sistema operativo inizializza i programmi, organizza l’utilizzo della CPU, e li termina quando hanno completato la loro esecuzione. Poiché molti programmi sono in esecuzione contemporaneamente nel sistema, il SO esegue lo *scheduling* per selezionare il programma da eseguire.

Gestione delle risorse

L'allocazione e la deallocazione delle risorse possono essere effettuate usando una tabella delle risorse. Ogni elemento della tabella contiene il nome e l'indirizzo di una risorsa (come può essere una stampante o un disco) e il suo stato attuale (libero o allocato).

Nome della risorsa	Classe	Indirizzo	Stato dell'allocazione
stampante1	Stampante	101	Allocata a P ₁
stampante2	Stampante	102	Libera
stampante3	Stampante	103	Libera
disco1	Disco	201	Allocata a P ₁
disco2	Disco	202	Allocata a P ₂
cdw1	CD writer	301	Libera

Tale tabella viene creata dalla procedura di boot rilevando la presenza di dispositivi di I/O nel sistema, e viene aggiornata dal SO in maniera costante.

Due sono le **strategie per l'allocazione delle risorse** più comuni:

- *partizionamento delle risorse*: Il SO decide a priori quali risorse allocare ad ogni programma utente; divide quindi le risorse in *partizioni* (collezione di risorse). In questo caso la tabella delle risorse contiene le entrate per le partizioni. Si tratta di un sistema semplice da implementare, ma privo di flessibilità.
- *Basata su gruppo (pool-based)*: Il SO consulta la tabelle delle risorse e, se la risorsa è libera, la alloca.
Tale strategia provoca un minor overhead nell'allocare e deallocare le risorse, e ne consente un uso più efficiente. (è utilizzata dai SO moderni)

Risorse virtuali

Una **risorsa virtuale** è una risorsa fittizia, ovvero è una illusione creata dal SO attraverso l'uso di risorse reali in modo da dare l'impressione di avere un numero di risorse maggiore di quelle effettivamente disponibili.

Un *server di stampa* è un tipico esempio di risorsa virtuale. Quando un programma desidera stampare un file, il server di stampa copia semplicemente il file nella coda di stampa,, mentre il programma continua la sua esecuzione come se la stampa fosse già avvenuta.

Inoltre, un SO può fornire una risorsa chiamata *memoria virtuale*, attraverso la quale si ha l'impressione di avere una memoria maggiore di quella fisicamente disponibile nel computer.

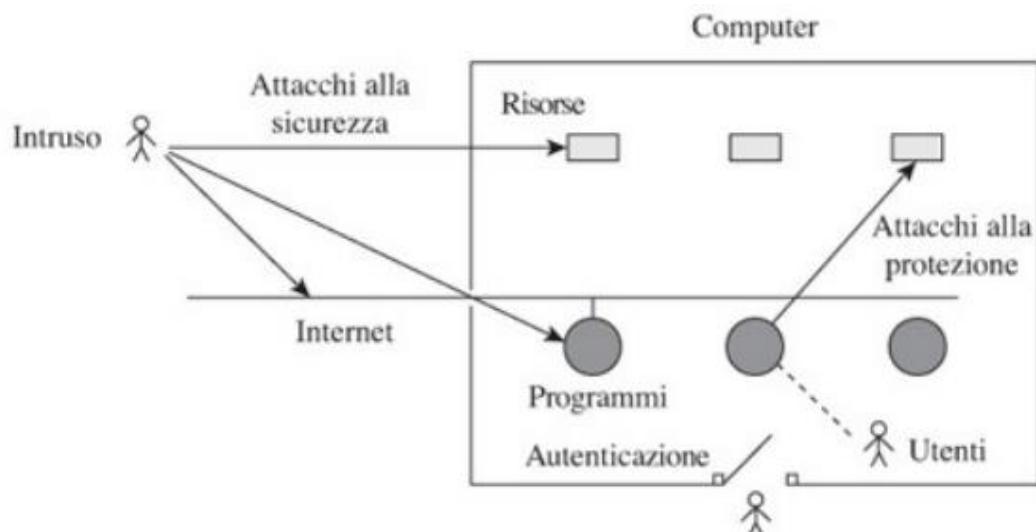
Alcuni SO creano *macchine virtuali* (VM) in modo che ogni macchina possa essere allocata ad un utente, così da eliminare le interferenze reciproche fra gli utenti e così che ogni utente possa utilizzare il suo SO preferito.

Sicurezza e protezione

un SO deve garantire che nessun utente possa utilizzare in maniera illegale programmi e risorse del sistema, o interferire con il loro funzionamento in alcun modo. La funzione della *sicurezza* riguarda l'uso illegale o le interferenze operate da persone o programmi fuori dal controllo del sistema operativo, mentre la funzione di *protezione* riguarda le stesse situazioni ma eseguite dai suoi utenti.

I sistemi operativi fronteggiano le minacce alla sicurezza attraverso una varietà di mezzi, utilizzando sofisticate tecniche di autenticazione o utilizzando *firewall* per filtrare il traffico di rete non autorizzato.

Ci si aspetta inoltre che anche gli utenti contribuiscano alla sicurezza utilizzando password difficile da indovinare ed essendo cauti nello scaricare programmi da internet.



LEZIONE 3 (1) – PANORAMICA SUI SO

AMBIENTI DI CALCOLO E NATURA DELLE ELABORAZIONI

Un ambiente di calcolo (o di elaborazione) consiste di un computer, delle sue interfacce con altri sistemi e dei servizi forniti dal suo SO ai propri utenti ed ai loro programmi.

Nel corso del tempo abbiamo assistito a diversi tipi di *ambienti di calcolo*, quali: non interattivi; interattivi; real-time, distribuiti ed embedded; moderni.

Ambienti di calcolo non interattivi

Rappresentano le forme più vecchie di ambienti di elaborazione. I questi ambienti l'utente non può intervenire nell'esecuzione durante l'elaborazione.

Il SO è orientato *all'uso efficiente* delle risorse e le *elaborazioni utilizzate* sono programmi o *job*.

Ambienti di calcolo interattivi

In questi ambienti l'utente può intervenire nell'esecuzione durante l'elaborazione, ad esempio inserendo dati o il nome di un file.

In questo caso il SO è orientato sulla *riduzione della quantità media di tempo* richiesto per implementare un'interazione tra l'utente e una sua elaborazione.

L'esecuzione di un programma è chiamata *processo*.

La richiesta di elaborazione di un utente ad un processo prende il nome di *sottorichiesta*.

Elaborazioni:

Elaborazione	Descrizione
Programma	Un <i>programma</i> è un insieme di funzioni o moduli, che includono alcune funzioni o moduli contenuti nelle librerie.
Job	Un <i>job</i> è una sequenza di programmi che insieme raggiungono un obiettivo comune. Non ha senso eseguire un programma in un job se il programma precedente del job non è stato eseguito con successo.
Processo	Un <i>processo</i> è l'esecuzione di un programma.
Sottorichiesta	Una <i>sottorichiesta</i> è una richiesta di elaborazione effettuata da un utente a un processo. Ogni sottorichiesta produce una singola risposta, che consiste di un insieme di risultati o azioni.

nb: per quanto riguarda un job, pensare alla fase di compilazione, linking ed esecuzione di un programma in C.

Ambienti real-time, distribuiti ed embedded

Si trattano di ambienti che hanno delle necessità speciali, per i quali sono stati sviluppati dei speciali ambienti di elaborazione.

Un ambiente **real-time** deve rispettare specifici *vincoli temporali*, per cui le sue azioni sono efficaci solo se completate in uno specifico intervallo di tempo.

Di ciò se ne occupa il SO utilizzando tecniche speciali.

Un ambiente **distribuito** consente di utilizzare le risorse presenti in diversi sistemi attraverso una rete.

Un ambiente **embedded** è un ambiente in cui il computer è una parte di uno specifico sistema HW.

Generalmente il computer ha una configurazione minimale ed è poco costoso, e il suo SO deve soddisfare i vincoli temporali derivanti dalla natura del sistema controllato (es. scheda di una lavatrice).

Ambienti moderni

Possiamo vedere gli ambienti moderni come un ambiente che ha preso il meglio dai suoi predecessori, in quanto ne implementa delle caratteristiche.

Di conseguenza, il SO deve adottare complesse strategie per gestire le elaborazioni degli utenti e le risorse; per esempio, deve ridurre il tempo medio richiesto per implementare l'interazione tra un utente e l'applicazione e anche assicurare un uso efficiente delle risorse.

EFFICIENZA, SERVIZIO PER L'UTENTE e PRESTAZIONI DEL SISTEMA

Valutare l'**EFFICIENZA** nell'uso di una risorsa equivale a vedere quanto la risorsa non è utilizzata o sprecata e, relativamente all'utilizzo della risorsa, quanto è stata produttiva.

Alcuni aspetti della **CONVENIENZA PER L'UTENTE** sono intangibili e dunque impossibili da misurare numericamente (come l'usabilità delle interfacce).

Altre invece sono misurabili, come il *servizio per l'utente*, che indica quanto velocemente un'elaborazione dell'utente è stata completata dal SO.

In particolar modo distinguiamo:

- **Tempo di turnaround:** il tempo trascorso dalla sottomissione da parte di un utente di un job, di un programma o di un processo fino al momento in cui i risultati sono restituiti all'utente;
- **Tempo di risposta:** il tempo trascorso dalla sottomissione di una sottorichiesta da parte di un utente fino al momento in cui il processo risponde a essa.

Stabilita la giusta combinazione di efficienza della CPU e servizio per l'utente, è importante poter misurare le **PRESTAZIONI** del SO. Essa indica il tasso con il quale il computer effettua il lavoro durante il suo funzionamento.

Le prestazioni del sistema sono caratterizzate dalla quantità di lavoro svolto per unità di tempo e sono generalmente misurate dal **throughput**, il quale è *definito* come:

Il numero di job, programmi o sottorichieste completati da un sistema in una unità di tempo.

Riassumendo:

Aspetto	Misura	Descrizione
Efficienza d'uso	Efficienza della CPU	Percentuale di utilizzo della CPU
	Efficienza della memoria	Percentuale di utilizzo della memoria
Prestazioni del sistema	Throughput	Quantità di lavoro svolto per unità di tempo
Servizio per l'utente	Tempo di turnaround	Tempo di completamento di un job o di un processo
	Tempo di risposta	Tempo di risposta a una sottorichiesta

CLASSI DI SISTEMI OPERATIVI

Le classi di sistemi operativi si sono evolute nel tempo con l'evoluzione dei computer e delle aspettative degli utenti.

Distinguiamo **cinque classi** fondamentali di sistemi operativi il cui nome rispecchia le loro caratteristiche peculiari:

Classe del SO	Periodo	Obiettivo principale	Concetti chiave
Elaborazione batch	anni '60	Tempo idle della CPU	Automatizzare la transizione tra i job
Multiprogrammazione	anni '60	Utilizzo delle risorse	Proprietà del programma, prelazione
Time-sharing	anni '70	Buon tempo di risposta	Scheduling, time slice, round-robin
Real time	anni '80	Rispettare i vincoli temporali	Scheduling real-time
Distribuiti	anni '90	Condivisione delle risorse	Controllo distribuito, trasparenza

L'hardware di elaborazione era costoso nei primi anni, pertanto l'*elaborazione batch* e i sistemi operativi *multiprogrammati* si focalizzavano sull'uso efficiente della CPU e delle altre risorse del computer.

Negli anni '70, l'hardware dei computer divenne più economico, così l'uso efficiente di un computer non fu più il primo obiettivo, che si spostò verso la produttività degli utenti.

Gli anni '80 videro l'avvento delle applicazioni real-time per il controllo o il tracciamento delle attività del mondo reale.

Negli anni '90, l'ulteriore calo dei prezzi dell'hardware condusse allo sviluppo dei sistemi distribuiti, grazie ai quali diversi computer condividevano le proprie risorse attraverso la rete.

1. SISTEMI DI ELABORAZIONE BATCH

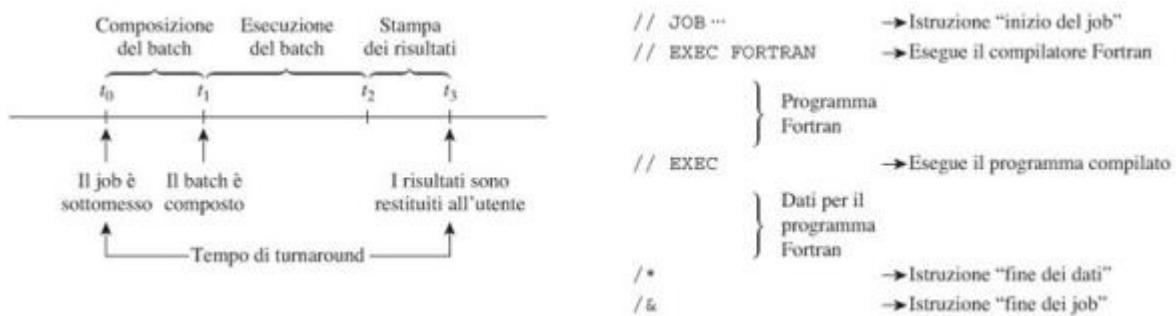
I computer negli anni '60 non erano interattivi e le schede perforate erano il mezzo di input principale. Dunque, un job e i suoi dati consistevano in gruppo di schede. Un operatore caricava le schede in un lettore per impostare l'esecuzione di un job. Questa azione causava una perdita di tempo prezioso di CPU; l'elaborazione batch fu introdotto per prevenire questo spreco.

Un **batch** è una sequenza di job utente assemblati per essere elaborati dal sistema operativo.

Quando l'operatore forniva un comando per iniziare l'elaborazione di un batch, il *batching kernel* impostava l'elaborazione del primo job del batch. Al termine del job, cominciava l'esecuzione del job seguente e così via (in tal modo, l'operatore doveva intervenire solo all'inizio e alla fine del batch).

I lettori di schede e le stampani rappresentavano un collo di bottiglia negli anni '60, così i sistemi di elaborazione batch utilizzarono il concetto di lettori di schede e stampanti virtuali.

Tempo di turnaround in un sistema di elaborazione batch (ed esempio job in Fortran):



2. SISTEMI MULTIPROGRAMMATI

I sistemi operativi multiprogrammati furono sviluppati per fornire un utilizzo efficiente delle risorse in un ambiente di elaborazione non interattivo, e prende il suo nome dal fatto che mantiene molti programmi utente in memoria.

Utilizza il DMA per le operazioni di I/O in modo da poter eseguire le operazioni di I/O di alcuni programmi mentre utilizza la CPU per eseguire altri programmi. Questa organizzazione fa un uso efficiente sia della CPU che dei dispositivi di I/O.

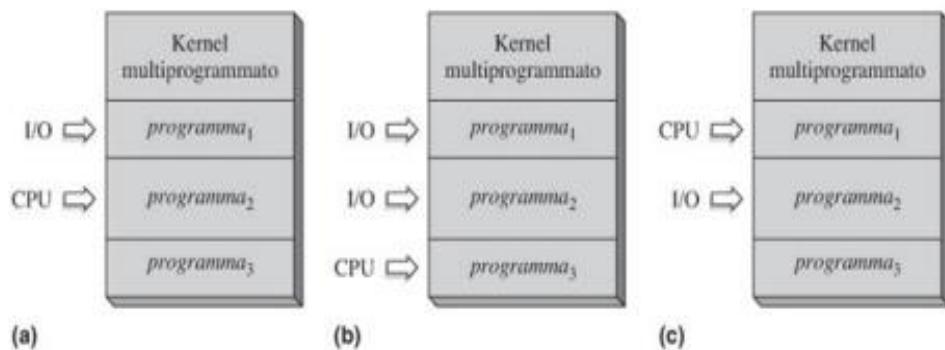
Il tempo di *turnaround* di un programma è la misura appropriata del servizio per l'utente in un sistema multiprogrammato.

Un sistema multiprogrammato deve possedere diverse **caratteristiche**:

Caratteristica	Descrizione
DMA	La CPU avvia un'operazione di I/O quando viene eseguita un'istruzione di I/O. Il DMA implementa il trasferimento dei dati senza coinvolgere la CPU e genera un interrupt di I/O quando il trasferimento è completato.
Protezione della memoria	Un programma può accedere solo alla parte della memoria definita dal contenuto del <i>registro base</i> e del <i>registro size</i> .
Modalità kernel e utente della CPU	Alcune istruzioni, chiamate <i>istruzioni privilegiate</i> , possono essere eseguite solo quando la CPU si trova in modalità kernel. Un interrupt di programma si verifica se un programma tenta di eseguire un'istruzione privilegiata quando la CPU è in modalità utente.

Funzionamento di un sistema multiprogrammato:

(a) *programma*₂ è in esecuzione; (b) *programma*₂ avvia un'operazione di I/O, e il *programma*₃ viene schedulato; (c)



l'operazione di I/O del *programma*₁ viene completata e il programma viene schedulato.

PRIORITÀ DEI PROGRAMMI

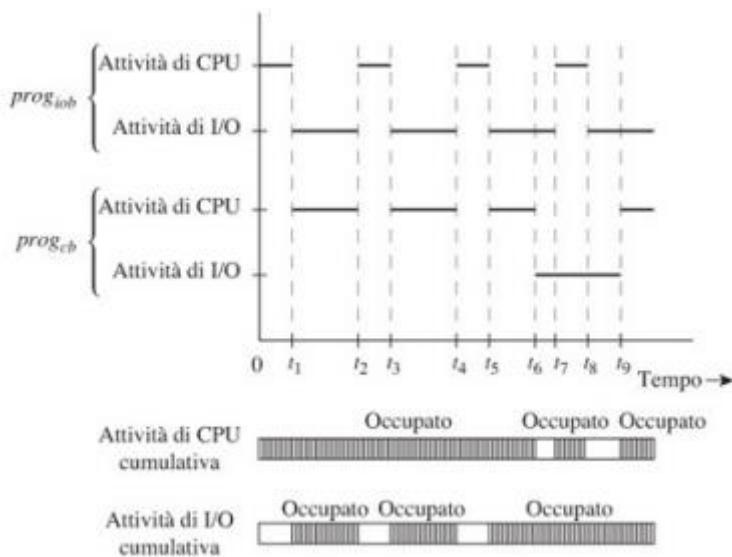
Una misura appropriata delle prestazioni di un SO multiprogrammato è il *throughput*, ovvero il rapporto del numero di programmi elaborati e il tempo totale impiegato per elaborarli.

Il SO mantiene sempre un numero sufficiente di programmi in memoria, in modo che la CPU e i dispositivi di I/O abbiano lavoro sufficiente da effettuare. Questo numero è denominato *grado di multiprogrammazione*. Tuttavia un grado elevato non può garantire un buon utilizzo sia della CPU che dei dispositivi di I/O.

Dunque il SO multiprogrammato adotta due **tecniche**:

- utilizza un appropriato **mix di programmi**, così alcuni dei programmi in memoria siano *programmi CPU-bound*, ovvero programmi che necessitano di molta elaborazione ma poche operazioni di I/O, e altri siano *programmi I/O-bound*, ovvero programmi che effettuano poca elaborazione ma molte operazioni di I/O.
- utilizza lo **scheduling a priorità di prelazione**, ovvero ad ogni programma viene assegnata una priorità. La CPU è sempre allocata al programma con la priorità più alta (se questo è pronto per l'esecuzione).
Un programma a più bassa priorità viene prelazione se uno con priorità più alta richiede l'uso della CPU.

Con **priorità** si intende un criterio mediante il quale lo scheduler decide quale richiesta debba essere schedelluata quando molte richieste sono in attesa di essere servite.



Esempio di priorità: prog_{lob} è un programma a maggior priorità rispetto a prog_{cb} , infatti noteremo che quando prog_{lob} ha bisogno di effettuare operazioni legate alla cpu avrà priorità rispetto a prog_{cb} , e quando invece la disimpegna (per esempio, per effettuare delle operazioni di I/O) la priorità passerà a prog_{cb} .

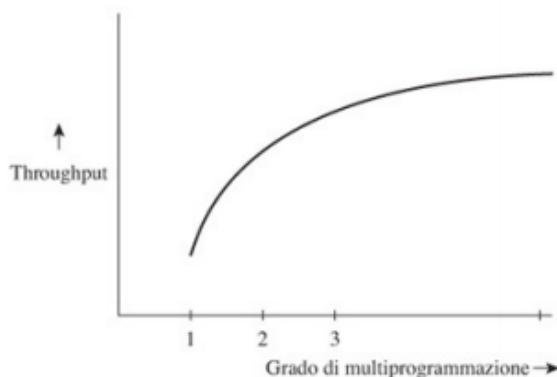
Notare che due programmi non possono utilizzare la stessa CPU contemporaneamente, tuttavia possono essere impegnati contemporaneamente in operazioni di I/O se tali dispositivi di I/O sono diversi.

Notare inoltre che se ci sono “spazi vuoti” potremmo inserire altri programmi che impegnano quegli spazi per avere un maggior grado di multiprogrammazione.

Quando si mantiene un appropriato mix di programmi, ci si può aspettare che un aumento del grado di multiprogrammazione risulti in un **aumento del throughput**.

Se abbiano un unico programma, allora il throughput sarà 1 in quanto tale programma avrà sempre la possibilità di utilizzare la CPU. Se inseriamo altri programmi a più bassa priorità, il loro contributo al throughput sarà limitato dall’opportunità di utilizzo della CPU.

Infatti, non è detto che all’aumento del numero di programmi aumenti necessariamente il throughput, in quanto se i programmi a più bassa priorità non hanno possibilità di essere eseguiti allora non possono contribuire all’aumento del throughput.



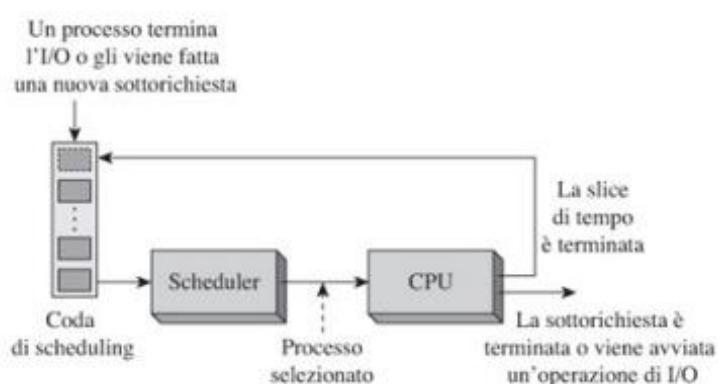
3. SISTEMI TIME-SHARING

Un sistema operativo time-sharing è progettato per fornire una risposta veloce alle sottorichieste effettuate dagli utenti. L’obiettivo è ottenuto condividendo il tempo di CPU tra i processi in modo tale che ogni processo al quale è stata effettuata una sottorichiesta ottenga un tempo della CPU senza attendere troppo.

La tecnica di scheduling utilizzata nei kernel time-sharing viene chiamata *scheduling round-robin con time-slicing*.

Ovvero, il kernel mantiene una cosa di scheduling dei processi, e tali processi vengono selezionati ed eseguiti nel solito modo. Tuttavia in questo caso, se il processo impiega per troppo tempo la CPU (ovvero scade lo *slice di tempo*), questo viene prelazionato e reinserito nella coda.

Esempio di scheduler round-robin con time-slice:



Con **time slice** si indica la più grande porzione di tempo di CPU che ogni processo può utilizzare prima che venga schedulato.

La misura appropriata per misurare il servizio per l'utente in un sistema time-sharing è il tempo utilizzato per eseguire una sottorichiesta, ovvero il tempo di risposta (rt).

$$rt = n \times (\delta + \sigma)$$
$$\eta = \delta / (\delta + \sigma)$$

Dove

n: numero utenti che usano il sistema,
 δ : tempo richiesto per completare una sotto-richiesta,
 σ : overhead dello scheduling,
 η : efficienza di CPU

Notare come il tempo di risposta effettivo potrebbe essere diverso poiché:

- alcuni utenti potrebbero essere inattivi;
- alcuni programmi possono richiedere più di δ secondi di CPU.

SWAPPING DEI PROGRAMMI

Lo swapping è una tecnica che rimuove temporaneamente un processo dalla memoria di un elaboratore, e viene utilizzata per servire un maggior numero di processi rispetto a quelli che possono essere effettivamente presenti in memoria.

Il kernel effettua un'operazione di *swap-out* su un processo che verosimilmente non verrà schedulato nel prossimo futuro, copiando le sue istruzioni e dati su disco.

Il kernel poi carica un altro processo in quest'area di memoria attraverso un'operazione di *swap-in*.

4. SISTEMI OPERATIVI REAL-TIME

Nelle applicazioni real-time, gli utenti hanno bisogno di un computer per eseguire determinate azioni tempestivamente, questo per controllare azioni in un sistema esterno o per prenderne parte. La puntualità dipende dai vincoli temporali.

Con **applicazione real-time** si intende un programma che risponde alle attività di un sistema esterno entro un intervallo di tempo massimo determinato dal sistema esterno.

Se l'applicazione impiega troppo tempo per rispondere ad un'attività, può verificarsi un fallimento nel sistema esterno. Usiamo il termine *requisito di risposta* di un sistema per indicare il più grande valore di tempo di risposta per il quale il sistema può funzionare correttamente.

SISTEMI REAL-TIME HARD e SOFT

Possiamo distinguere due tipologie di sistemi real-time:

Un sistema **real-time hard** soddisfa i requisiti di risposta in ogni condizione, e tipicamente è dedicato all'elaborazione di applicazioni real-time.

Un sistema **real-time soft** fa del suo meglio per soddisfare il requisito di risposta di un'applicazione real-time ma non può garantire che lo farà in ogni condizione.

Tipicamente, soddisfa tali requisiti in maniera probabilistica (si pensi alle applicazioni multimediali: un video può peggiorare ma comunque essere usufruito).

Caratteristiche di un SO real-time:

Caratteristica	Spiegazione
Concorrenza all'interno di un'applicazione	Un programmatore può indicare che alcune parti di un'applicazione debbano essere eseguite in maniera concorrente l'una con l'altra. Il SO considera l'esecuzione di ognuna di queste parti come un processo.
Priorità del processo	Un programmatore può assegnare delle priorità al processo.
Scheduling	Il SO utilizza politiche di scheduling basate su priorità o su deadline.
Eventi specifici per il dominio, interrupt	Un programmatore può definire speciali situazioni nel sistema esterno come eventi, associare a essi degli interrupt e specificare azioni per la gestione di questi eventi.
Predicibilità	Le politiche e l'overhead del SO dovrebbero essere predibili.
Affidabilità	Il SO garantisce che un'applicazione possa continuare a funzionare anche quando si verificano malfunzionamenti nel computer.

5. SISTEMI OPERATIVI DISTRIBUITI

Un sistema operativo distribuito si compone di diversi computer singoli connessi attraverso una rete. Ogni computer potrebbe essere un PC, un sistema multiprocessore o un cluster.

In tal modo, in un sistema distribuito esistono molte risorse di un determinato tipo (memorie, CPU, dispositivi di I/O). Un sistema operativo distribuito sfrutta la molteplicità delle risorse e la presenza di una rete per fornire *diversi benefici*. Tuttavia, la possibilità di malfunzionamenti di rete o di singoli computer complica il funzionamento del sistema operativo e necessita di speciali tecniche di implementazione.

Tali sistemi presentano diversi benefici:

Beneficio	Descrizione
Condivisione delle risorse	Le risorse possono essere utilizzate attraverso i limiti dei singoli sistemi.
Affidabilità	Il SO continua a funzionare anche quando i computer o le risorse in esso contenute non funzionano.
Incremento della velocità di elaborazione	I processi di un'applicazione possono essere eseguiti su differenti sistemi per velocizzarne il completamento.
Comunicazione	Gli utenti possono comunicare tra di loro a prescindere dalle loro posizioni nel sistema.

Formalmente, un *sistema distribuito* è un sistema composto di due o più nodi, in cui ogni nodo è un computer con un proprio clock e una propria memoria, dell'hardware di rete con la capacità di effettuare alcune funzioni di controllo di un SO.

Possiamo distinguere 3 **concetti/tecniche** utilizzate in un SO distribuito:

Concetto/tecnica	Descrizione
Controllo distribuito	Una funzione di controllo viene effettuata attraverso la collaborazione di diversi nodi, probabilmente <i>tutti</i> i nodi, in un sistema distribuito.
Trasparenza	Si può accedere a una risorsa o a un servizio senza dover conoscere la sua posizione nel sistema distribuito.
Remote procedure call (RPC)	Un processo chiama una procedura che è messa a disposizione da un computer remoto. La RPC è analoga a una procedura o una chiamata di funzione in un linguaggio di programmazione, fatta eccezione per il fatto che il SO passa i parametri alla procedura remota attraverso la rete e restituisce i risultati attraverso la rete.

MODERNI SISTEMI OPERATIVI

Gli utenti sono coinvolti in diverse attività in un moderno ambiente di elaborazione. Dunque, un moderno sistema operativo non può usare una strategia uniforme per tutti i processi; invece, deve usare una strategia appropriata per ogni singolo processo.

Pertanto, il SO utilizzerà tecniche diverse per situazioni diverse:

Concetto	Tipico esempio d'uso
Elaborazione batch	Evita perdite di tempo dovute all'inizializzazione per ogni utilizzo di una risorsa; per esempio, le transazioni su database sono elaborate in modalità batch nelle elaborazioni di ufficio e le elaborazioni scientifiche sono eseguite in modalità batch nelle organizzazioni di ricerca e nei laboratori clinici.
Scheduling a priorità con prelazione	Favoriscono le applicazioni ad alta priorità e consentono un uso efficiente delle risorse assegnando alte priorità ai processi interattivi e basse priorità ai processi non interattivi.
Time-slicing	Per prevenire che un processo monopolizzi la CPU; aiuta a fornire buoni tempi di risposta.
Swapping	Incrementa il numero di processi che possono essere serviti simultaneamente; consente di migliorare le prestazioni del sistema e i tempi di risposta dei processi.
Creazione di processi multipli in una applicazione	Permette di ridurre la durata di un'applicazione; è più efficace quando l'applicazione contiene sostanziali attività di CPU e I/O.
Condivisione delle risorse	Consente di condividere risorse come stampanti laser o servizi come file server in un ambiente LAN.

LEZIONE 3 (2) – STRUTTURA DEI SO

FUNZIONAMENTO DI UN SO

Quando un computer è avviato, è eseguita una procedura di *boot*, la quale analizza la sua configurazione, ovvero il tipo di CPU, la dimensione della memoria, i dispositivi di I/O ed altri dettagli hardware.

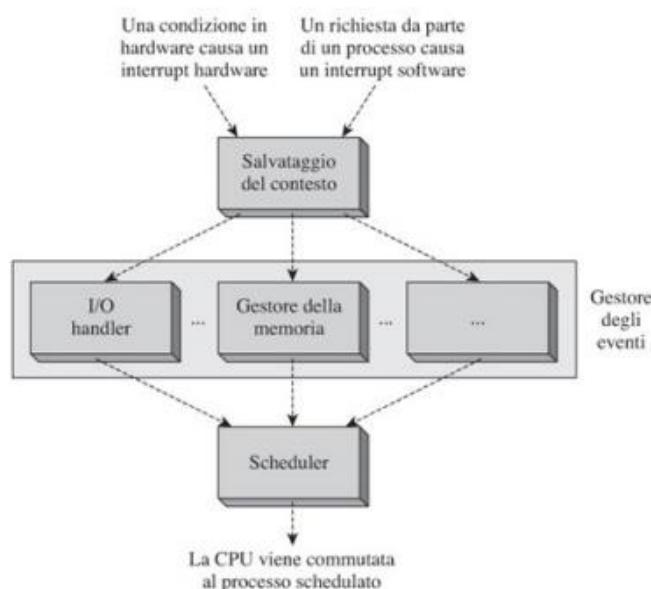
Inoltre carica parte del SO in memoria, inizializza le strutture dati e passa ad esso il controllo del sistema.

Durante il funzionamento del computer, possono verificarsi delle interruzioni causate da eventi (come il completamento di un'operazione di I/O o la terminazione di una time/slice) oppure da una chiamata di sistema fatta da un processo (*interruzione software*).

Il verificarsi di un'interruzione viene *gestita* da una routine di servizio, la quale esegue il salvataggio del contesto ed attiva il gestore degli eventi.

Lo scheduler seleziona il processo da servire.

Visione di insieme del funzionamento di un SO:



Le funzioni svolte dal SO sono:

Funzione	Descrizione
Gestione dei processi	Avvio e terminazione dei processi, scheduling.
Gestione della memoria	Allocazione e deallocazione della memoria, swapping, gestione della memoria virtuale.
Gestione dell'I/O	Servizio degli interrupt di I/O, avvio delle operazioni di I/O, ottimizzazione delle prestazioni dei dispositivi di I/O.
Gestione dei file	Creazione, memorizzazione e accesso ai file.
Sicurezza e protezione	Prevenzione dalle interferenze tra processi e risorse.
Gestione delle comunicazioni (network)	Inviare e ricevere dati attraverso la rete

POLITICHE E MECCANISMI

Nel determinare come il SO esegue una funzione, il progettista deve pensare a **due livelli** distinti:

- **Politica:** principio in base al quale il SO esegue una funzione (decide quindi cosa deve essere fatto);
- **Meccanismo:** azione necessaria per implementare una politica (determina quindi come deve essere fatta e la attua).

Ad esempio, lo scheduling di round-robin è una politica, mentre mantenere una cosa dei processi pronti e commutare la CPU per eseguire il processo selezionato (dispatching) è il suo meccanismo.

PORATABILITÀ ed ESPANDIBILITÀ DEI SO

La portabilità e l'espandibilità sono due aspetti importanti per un SO che vanno tenuti bene in mente.

Portabilità, di cui distinguiamo nel dettaglio:

- **Porting:** si intende adattare il software per usarlo in un nuovo sistema di computer;
- **Portabilità:** si intende la semplicità con cui un programma può essere sottoposto a porting. Maggiore sarà la portabilità, minore sarà lo sforzo del porting.
- **Porting di un SO:** riguarda cambiare parti del suo codice che dipendono dall'architettura per funzionare con il nuovo HW.
Quando si sviluppa un SO, sarebbe ideale avere quante meno parti possibili che dipendono dall'HW così da aumentare la sua portabilità.

Estendibilità: ovvero la facilità con cui possono aggiungere nuove funzionalità ad un sistema software. Tale estendibilità è necessaria per due scopi:

- Incorporare nuovo HW in un sistema, come nuovi dispositivi di I/O o schede di rete;
- Fornire nuove caratteristiche per soddisfare nuove pretese degli utenti.

I primi SO non fornivano alcun tipo di estendibilità, mentre quelli moderni facilitano l'aggiunta di un driver di dispositivo.

Ancor meglio, tali SO forniscono anche capacità *plug-and-play*, ovvero se viene inserito un nuovo dispositivo di I/O durante l'esecuzione del sistema, questo viene intercettato dal meccanismo delle interruzioni (e ne viene caricato il driver).

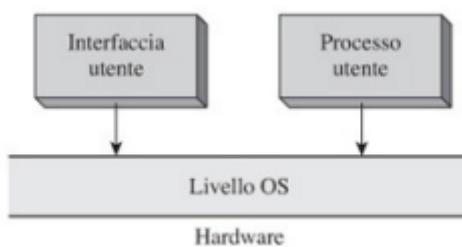
I Sistemi Operativi possono essere progettati in vari modi, e quindi ne distingiamo di diversi tipi: *SO con struttura monolitica*, *SO con struttura a strati*, *SO basati su kernel* e *SO basati su micro-kernel*.

SO CON STRUTTURA MONOLITICA

I primi SO avevano una struttura monolitica, ovvero il SO costituiva un singolo strato software tra l'utente e la nuda macchina (hardware).

Questa tipologia di SO ha diversi **problemi**: sono difficili da *implementare*, *estendere* ed hanno *cattiva portabilità* (in quanto il codice dipende dalla macchina sui cui è distribuito il SO). Inoltre rende il *testing* e il *debugging* *difficoltoso*, comportando elevati costi di manutenzione e potenziamento.

I **vantaggi** sono rappresentati dall'elevata efficienza di comunicazioni fra i vari moduli e con la macchina sottostante.



SO CON STRUTTURA A STRATI

Il SO è diviso in un numero di strati (livelli), ciascuno costruito in cima agli strati inferiori. Lo strato di fondo (strato 0) è l'HW, mentre quello in cima (strato N) è l'interfaccia utente.

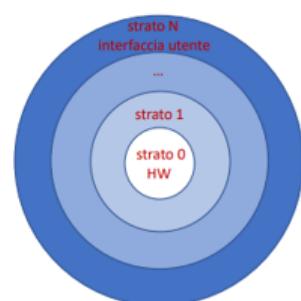
Con la modularità, gli strati sono selezionati in modo tale che ognuno usa le funzioni (operazioni) e i servizi dei soli strati inferiori.

Tale modularità rappresenta un **vantaggio** in quanto se si presenta un errore in un certo strato, dovrà controllare solo quello strato.

Di conseguenza, il debugging e l'implementazione risultano essere più semplici.

Strati nel sistema multi-programmato THE

Layer	Description
Layer 0	Processor allocation and multiprogramming
Layer 1	Memory and drum management
Layer 2	Operator-process communication
Layer 3	I/O management
Layer 4	User processes



Tuttavia, i SO con struttura a strati hanno diversi **svantaggi**:

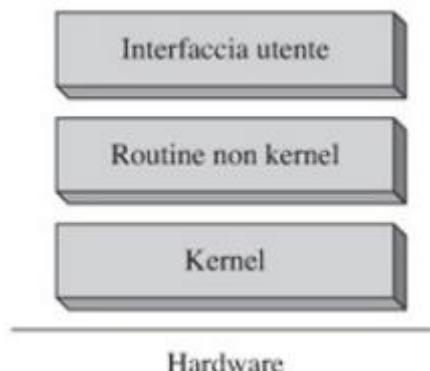
- risultano essere *più lenti*, perché per eseguire una determinata funzione potrei dover attraversare più strati;
- *Progettazione complessa*, infatti sviluppare un SO a strati non è banale;
- *Limitata estendibilità*.

SO BASATI SU KERNEL

I SO basati su kernel nascono dal bisogno di una maggior portabilità e convenienza nella progettazione del SO, e nella codifica delle routine non kernel.

Il *kernel* è il cuore di un SO e fornisce un insieme di istruzioni e servizi per supportare differenti funzioni. Il resto del SO è organizzato come un insieme di *routine non kernel*.

Se il kernel è piccolo (cioè costituito da poche funzioni) potrebbe convenire implementarlo in modo monolitico così da aumentarne l'efficienza (fu il caso di UNIX negli anni '70).



Un SO così costituito fornisce diverse **funzionalità**:

Funzionalità del SO	Esempi di funzioni e servizi del kernel
Gestione dei processi	Salva il contesto del programma interrotto, effettua il dispatch di un processo, manipola le liste di scheduling.
Comunicazioni tra processi	Invia e riceve messaggi dai processi.
Gestione della memoria	Imposta le informazioni relative alla protezione della memoria, effettua lo swap-in e lo swap-out, gestisce i page fault (ovvero l'interrupt "non presente in memoria", detto missing from memory, del Paragrafo 1.4).
Gestione dell'I/O	Avvia le operazioni di I/O, elabora l'interrupt generato dal completamento di un'operazione di I/O, effettua il recupero da errori di I/O.
Gestione dei file	Apre un file, legge/scrive dati.
Sicurezza e protezione	Aggiunge informazioni di autenticazione per un nuovo utente, conserva le informazioni per la protezione dei file.
Gestione della rete	Invia/riceve dati attraverso messaggi.

Notare che un SO di questo tipo può comunque soffrire di alcuni **problemi**, che dipendono anche dalla stratificazione. Infatti può capitare che un comando del SO contenga una parte che dipende dall'architettura, e quindi si devono nuove creare nuove chiamate per arginare tale problema (generando overhead).

Per tal motivo, alcuni SO hanno pensato di inserire all'interno del kernel porzioni di codice che non dipendono dall'architettura.

Tuttavia, col corso del tempo si è andato ad inserire nel kernel sempre più porzioni di codice *non* kernel, rendendo il SO meno portabile è più pesante.

Per tal motivo, furono introdotti **moduli kernel** i quali potevano venir caricati dinamicamente, e i quali interagivano fra di loro con delle opportune interfacce.

Il kernel di base viene caricato durante il boot, mentre gli altri moduli venivano caricati solo quando necessario (ciò comporta anche un risparmio di memoria).

Tale strategia viene anche adottata per implementare i driver di dispositivo e nuove chiamate di sistema.

Visto che un modulo in modalità kernel può essere solo eseguito in modalità kernel, furono progettati **driver di dispositivo a livello utente** così da facilitare lo sviluppo, il debugging, la distribuzione e la robustezza.

Le prestazioni sono assicurate attraverso mezzi HW e SW.

SO BASATI SU MICRO-KERNEL

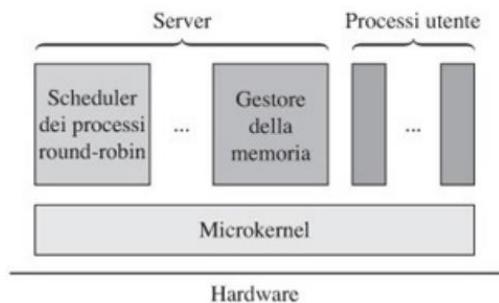
I micro-kernel furono sviluppati nei primi anni '90 per ovviare ai problemi di portabilità, estendibilità e affidabilità dei kernel.

Un micro-kernel è un nucleo essenziale del codice del SO:

- contiene solo un sottoinsieme dei meccanismi inclusi tipicamente nel kernel;
- supporta solo un piccolo numero di chiamate di sistema, usate e testate massicciamente;
- al di fuori del kernel, c'è meno codice essenziale.

Quindi si fa un passo indietro, rendendone nuovamente piccolo il kernel e aumentando la portabilità, l'estendibilità e la modificabilità.

Notare come il micro-kernel non include lo scheduler e il gestore della memoria, che invece sono eseguiti come server.



nb: i sistemi moderni utilizzano un mix di tutte le strategie descritte, e ciò è anche prevedibile in quanto ogni strategia ha i propri vantaggi e svantaggi.

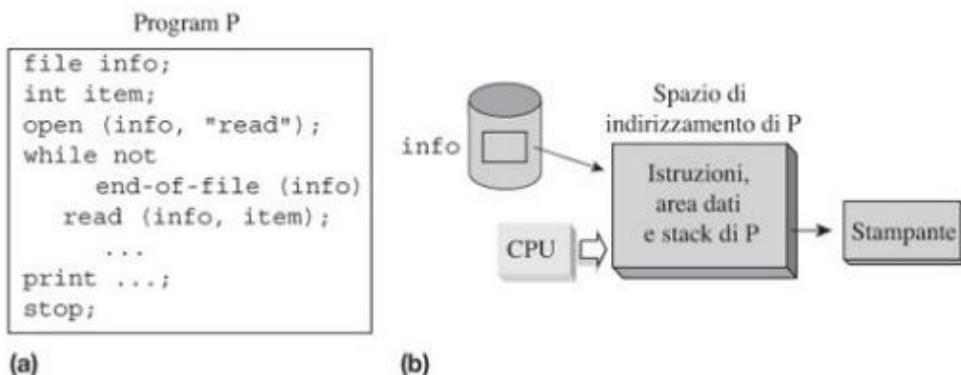
LEZIONE 4 – PROCESSI

Un processo è *un programma in esecuzione che utilizza un insieme di risorse*.

Si differenzia dal programma in quanto quest'ultimo è un'entità passiva che non effettua alcuna operazione da solo; quando viene eseguito, si parla di processo.

Quando un programma viene eseguito, il SO si occupa di: creare un processo, allocare a tale processo la quantità di memoria necessaria alla corretta esecuzione del programma stesso, e le risorse associate (come un file, una “stampante” o la CPU (che ovviamente non sarà sempre disponibile)).

Lo **spazio di indirizzamento** del processo conterrà le istruzioni, i dati e lo stack del processo stesso.



Un processo comprende sei **componenti**:

- **Id:** identificativo assegnato dal SO;
- **Codice:** è il codice del programma;
- **Dati:** dati usati durante l'esecuzione (inclusi quelli contenuti dai file usati dal programma);
- **Stack:** contiene parametri delle funzioni e procedure invocate durante l'esecuzione e i loro indirizzi di ritorno;
- **Risorse:** è l'insieme di risorse allocate dal SO;
- **Stato della CPU:** composto da PSW e GPR.

Ricordiamo che il PSW (Program Status Word) è un'area di memoria che contiene informazioni sullo stato dei programmi in esecuzione. Più nello specifico, è un insieme di flag presenti nella CPU che indicano lo stato di diversi risultati di operazioni matematiche.

I GPR (General Purpose Register) sono usati per far calcoli sui dati e conservare indirizzi.

Program counter (PC)	Condition code (CC)	Mode (M)	Memory protection information (MPI)	Interrupt mask (IM)	Interrupt code (IC)
-------------------------	------------------------	-------------	--	------------------------	------------------------

Un **programma** è un insieme di funzioni e procedure: le funzioni possono essere processi separati o possono costituire la parte di codice di un singolo processo.

Possiamo distinguere due tipi di **relazioni** fra programmi e processi:

Relazione	Esempi
Uno a uno	Una singola esecuzione di un programma sequenziale.
Molti a uno	Molte esecuzioni simultanee di un programma, esecuzione di un programma concorrente.

nb: con Molti a uno possiamo distinguere 2 casi: posso mandare in esecuzione più copie (istanze) del programma (è il caso dei client-server, dove ho n client che eseguono tutti lo stesso codice), oppure possono esserci istruzioni all'interno di un programma che demandano l'esecuzione di alcune funzioni ad altri processi.

I processi che coesistono nel sistema allo stesso momento sono detti *processi concorrenti*. Questi possono condividere il codice, i dati e le risorse con altri processi, ed inoltre hanno l'opportunità di interagire l'uno con l'altro durante la loro esecuzione.

PROCESSI E FIGLI

Quando un processo crea un altro processo, si dice che crea un **processo figlio**.

Il processo figlio avrà tutte le caratteristiche di un normale processo, ma *in più* conserverà l'ID del processo padre così da sapere a quale processo fa riferimento.

Un processo figlio potrà avere a sua volta altri processi figli.

L'insieme dei processi figli e genitori formano un *albero dei processi*.

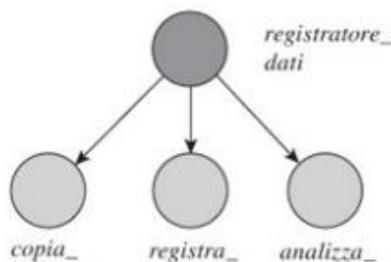
Il processo padre attribuisce parte del proprio lavoro a ciascuno dei propri figli, si parla quindi di **multi-tasking** all'interno di un'applicazione (indipendentemente se si ha un sistema mono o multi processore).

I processi figli hanno diversi **benefici**:

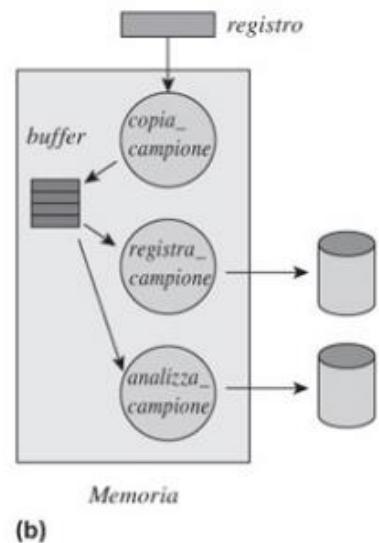
Beneficio	Spiegazione
Speedup dell'elaborazione	Azioni che il processo primario di un'applicazione avrebbe dovuto effettuare sequenzialmente se non avesse creato i processi figli, sarebbero effettuate concorrentemente quando crea i processi figli. Questo può ridurre la durata dell'applicazione, ovvero il tempo di esecuzione.
Priorità per le funzioni critiche	Un processo figlio che effettua una funzione critica può ottenere una priorità alta; questo può aiutare a soddisfare i requisiti real-time di un'applicazione.
Proteggere un processo genitore dagli errori	Il kernel termina un processo figlio se si verifica un errore durante la sua esecuzione. Il processo genitore non risente dell'errore e può essere in grado di eseguire un'azione di recupero.

Esempio di processo:

Il dati arrivano con una certa frequenza dal satellite (registro). *copia_campione* è il processo con criticità più alta perché da lui dipendono gli altri due processi, ed è anche quello con priorità più alta. Il campione viene inserito nel buffer, il quale sarà poi disponibile agli altri due processi.



(a)



(b)

CONCORRENZA E PARALLELISMO

Il **parallelismo** si riferisce alla caratteristica di verificarsi allo stesso momento, ottenuta utilizzando CPU multiple.

Due attività sono parallele se sono eseguite allo stesso momento.

La **concorrenza** è un'illusione di parallelismo, ottenuta in un SO alterando le operazioni di vari processi sulla CPU (la quale può gestire un solo processo per volta).

Due attività sono concorrenti se c'è un'illusione che esse sono eseguite in parallelo laddove solo una di loro può essere eseguita in un dato momento.

Sia la concorrenza che il parallelismo possono fornire un **miglior throughput**.

IMPLEMENTAZIONE

Per un SO, un processo è un'unità di lavoro computazionale. Il compito primario del kernel è controllare le operazione dei processi per assicurarne un uso efficace.

Il kernel viene attivato quando un *evento* porta alla generazione di un'interrupt. A tal punto il kernel effettua quattro funzioni:

1. *salvataggio del contesto*: salvare lo stato della CPU e le informazioni riguardanti le risorse del processo la cui esecuzione è stata interrotta;
2. *Gestione dell'evento*: analizzare la condizione che ha portato a un interrupt, o la richiesta da parte di un processo che ha portato a una system call;
3. *Scheduling*: selezionare il prossimo processo da eseguire sulla CPU;
4. *Dispatching*: impostare l'accesso alle risorse per il processo schedulato e caricare il suo stato salvato della CPU per iniziare o proseguire l'esecuzione.

Infine, avvive l'*uscita* dal kernel.

STATO DI UN PROCESSO e TRANSAZIONI DI STATO

Per poter gestire opportunamente i vari processi, è stato introdotto il concetto di **stato di un processo**, il quale è un indicatore che descrive la natura dell'attività corrente di un processo:

Stato	Descrizione
<i>Running</i>	Una CPU sta eseguendo le istruzioni di un processo.
<i>Blocked</i>	Il processo deve aspettare finché non viene soddisfatta una sua richiesta per una risorsa o finché non si verifica uno specifico evento.
<i>Ready</i>	Il processo richiede l'uso della CPU per continuare la sua esecuzione; tuttavia, non è stato ancora eseguito il dispatch.
<i>Terminated</i>	L'esecuzione del processo, ovvero, l'istanza del programma che rappresenta, è stata completata correttamente o è stata terminata dal kernel.

Una **transazione di stato** per un processo è un cambio del proprio stato, la quale può essere causata dall'occorrenza di qualche evento come l'inizio o la fine di un'operazione di I/O.

Partendo da un determinato stato, un processo può transire in altri stati se si verificano determinate situazioni:

- **Running**: andrà in quello *Terminated* se è stato completato o abortito.
Andrà in quello *Blocked* nel caso richieda l'uso di una risorsa o attenda che si verifichi un evento.
Andrà in quello *Ready* se viene prelazionato (viene schedulato un processo a più alta priorità, o perché la time-slice del processo è terminata).
- **Blocked**: Se la richiesta fatta dal processo è soddisfatta o si verifica un evento di cui il processo era in attesa, allora esso andrà nello stato *Ready*, così da informare lo scheduler che tale processo è pronto ad essere schedulato.
- **Ready**: Se il processo viene selezionato dallo scheduler e viene effettuato il dispatching, allora esso andrà nello stato *Running*.

esempio:



Un sistema time-sharing ha due processi P1 e P2

- Time slice= 10ms
- P1 -> CPU burst 15 ms e operazione di I/O 100 ms
- P2 -> CPU burst 30 ms e operazione di I/O di 60 ms

Temo	Evento	Considerazioni	Nuovo stato	
			P1	P2
0		P1 è schedulato	running	ready
10	P1 è prelazionato	P2 è schedulato	ready	running
20	P2 è prelazionato	P1 è schedulato	running	ready
25	P1 avvia I/O	P2 è schedulato	blocked	running
35	P2 è prelazionato	–	blocked	ready
		P2 è schedulato	blocked	running
45	P2 avvia I/O	–	blocked	blocked

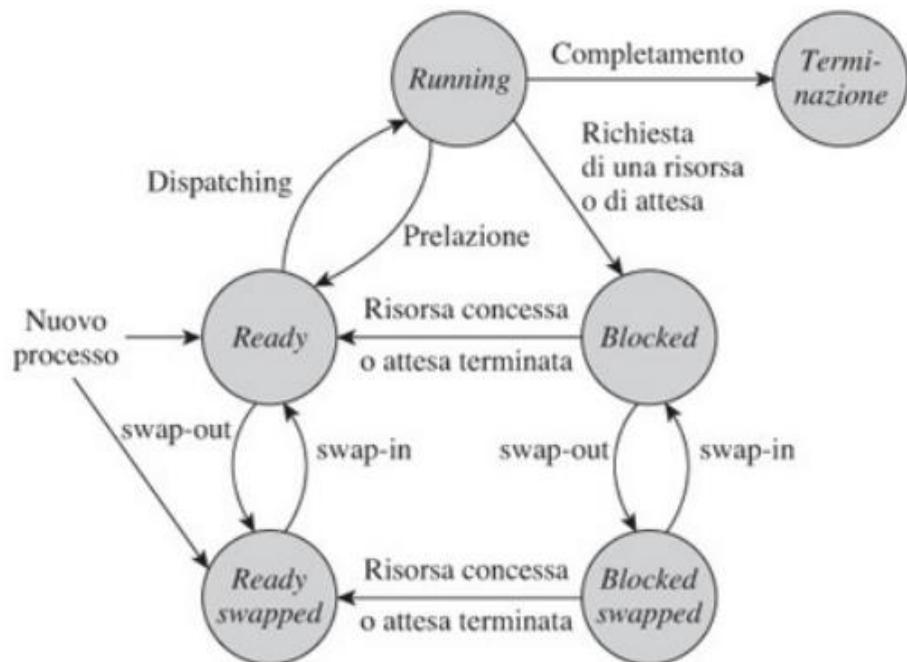
Un processo potrebbe non trovarsi in uno di questi stati fondamentali, per tal motivo il kernel deve prevedere **stati addizionali** per descrivere interamente la natura dell'attività di un processo.

Infatti ricordiamo che se un processo è stato swappato su disco, questo deve essere riportato in memoria prima che se ne possa riprendere l'esecuzione.

Tale processo prenderà il nome di **processo sospeso**, e distinguiamo due nuovi stati: *Ready-swapped* e *Blocked-swapped*.

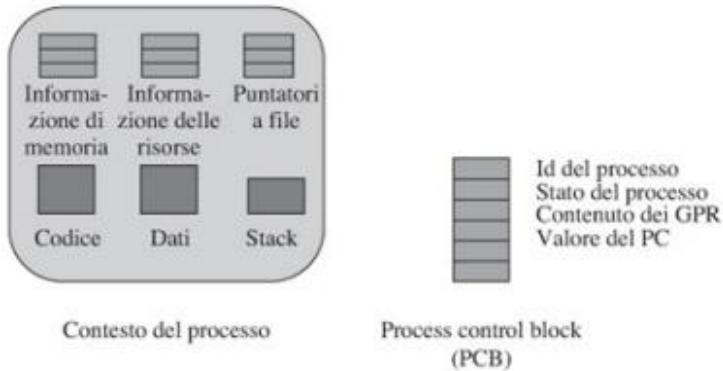
Quando viene creato un nuovo processo, questo può essere impostato a Ready o a Ready-swapped in base alla disponibilità di memoria.

Si può passare dallo stato "normale" a quello swapped e viceversa attraverso delle operazioni di *swap-in* e *swap-out*, ma non è possibile passare direttamente dallo stato *Blocked-swapped* a quello *Ready* o dallo stato *Ready-swapped* a quello *Blocked*.



CONTESTO DI UN PROCESSO e PROCESS CONTROL BLOCK

Come sappiamo, il kernel alloca le risorse al processo e lo schedula per l'utilizzo della CPU. Di conseguenza, il kernel vede il processo composto da due parti, le quali sono contenute nel *contesto del processo* e nel *process control block (PCB)*.



Contesto del processo

Il contesto del processo si compone di diverse parti:

- **Spazio di indirizzamento del processo**: componenti del processo relativi a codice, dati e stack;
- **Informazioni di allocazione di memoria**: le aree di memoria allocate al processo;
- **Stato delle attività sui file**: info sui file in uso;
- **Informazioni sulle interazioni del processo**: info necessarie per controllare le interazioni del processo con altri processi;
- **Informazioni sulle risorse**: info sulle risorse allocate al processo;
- **Informazioni di miscellanea**: altre informazioni necessarie per il funzionamento del processo.

PCB

Contiene informazioni riguardanti l'esecuzione dei programmi, di cui possiamo distinguere tre tipologie:

- **Info di identificazione**: Id del processo, Id del processo genitore e l'Id dell'utente che l'ha creato;
- **Informazioni di stato del processo**: il suo stato e il contenuto del PSW e GPR;
- **Informazioni per il controllo delle operazioni**: la sua priorità e le sue interazioni con altri processi.

Da notare inoltre che nel PCB è anche presente un campo puntatore che il kernel utilizza per la lista di PCB per lo scheduling, ovvero una coda di processi ready.

SALVATAGGIO DEL CONTESTO, SCHEDULING e DISPATCHING

Al verificarsi di un evento, vengo svolte diverse funzioni:

- la funzione di **salvataggio del contesto** salva lo stato della CPU del processo interrotto nel PCB e salva informazioni riguardo al contesto.
Inoltre, cambia lo stato del processo da *running* a *ready*;
- la funzione di **scheduling** usa le informazioni sullo stato salvate nel PCB per selezionare un processo ready per l'esecuzione e passa il suo id alla funzione di dispatching;
- la funzione di **dispatching** imposta il contesto del processo elezionato, cambia il suo stato a *running* e carica lo stato della CPU salvato dal PCB nella CPU.
Inoltre, scarica i buffer di traduzione dell'indirizzo usati dalla MMU, così da evitare problemi di protezione.

Supponiamo di avere due processi P1 e P2, dove la priorità di P2 > P1.

Supponiamo che P2 sia *bloccato* da un'operazione di I/O e che P1 vada in esecuzione.

Quando l'I/O di P2 sarà completata verranno effettuate una serie di operazioni che costituiscono la **commutazione** tra i due processi, cioè:

- avviene il salvataggio del contesto per P1 e il suo stato passa a *ready* (e P2 passa da bloccato a pronto);
- viene schedulato P2;
- avviene il dispatch di P2.

La *commutazione* avviene anche quando un processo in esecuzione diviene *bloccato* in conseguenza di una richiesta o della prelazione al termine di un time-slice.

L'occorrenza di un evento **non comporta la commutazione se** (1) causa una transizione di stato solo in un processo la cui priorità è inferiore a quella del processo la cui esecuzione è stata interrotta dall'evento oppure (2) non causa nessuna transizione di stato, per esempio se l'evento è causato da una richiesta che è immediatamente soddisfatta.

Chiaramente, la commutazione di un processo comporta **overhead**, il quale dipende dalle informazioni sullo stato delle processi, e il quale può essere ridotto attraverso istruzioni speciali che utilizzano alcune architetture di computer.

Da notare che la commutazione può comportare anche un overhead indiretto, in quanto il nuovo processo potrebbe non avere alcuna parte del suo spazio di indirizzamento nella cache, il quale caricamento comporta per appunto ulteriore overhead.

GESTIONE DEGLI EVENTI

Durante il funzionamento di un SO, possono occorrere vari eventi:

1. Evento per la creazione di un processo;
2. Evento per la terminazione di un processo;
3. Evento timer;
4. Evento per la richiesta di una risorsa;
5. Evento per il rilascio di una risorsa;
6. Evento per la richiesta di invio I/O;
7. Evento per il completamento di I/O;
8. Evento per l'invio di un messaggio;
9. Evento per la ricezione di un messaggio;
10. Evento per l'invio di un segnale;
11. Evento per la ricezione di un segnale;
12. Evento generato da interrupt da programma;
13. Evento per il malfunzionamento dell'hardware.

Eventi: creazione di processi

Si trattano di eventi si presentano quando un **utente invoca un comando** o un **processo richiede di creare un processo figlio** per eseguire un programma.

In entrambi i casi viene invocata una System call la quale crea il processo, e la routine predisposta alla gestione di questo evento si occupa di creare un PCB per il nuovo processo.

Eventualmente, dal kernel possono essere associate altre risorse standard.

Eventi: terminazione di processi

Si trattano di eventi che invocano una System call per la terminazione di se stessi o di processi figli.

La terminazione non è immediata ma si attende che siano completate le operazioni di I/O. In seguito viene rilasciata la memoria e le informazioni allocate.

Lo stato del processo è cambiato in *terminated*, e il suo PCB non viene rimosso finché il processo padre non ne preleva il suo stato di terminazione.

Nel caso fosse l'utente ad aver terminato il processo, dovrà essere il kernel a sollecitare il padre a prelevarne lo stato di terminazione.

Eventi: prelazione di processi

Si trattano di eventi che si presentano quando un processo nello stato *running* deve essere prelazionato se scade il suo time-slice.

La funzione di *salvataggio del contesto* ne cambia lo stato in *ready* e poi viene chiamato il gestore dell'evento *timer interrupt*, il quale si occuperà di spostare il solo PCB del processo in un'opportuna lista di scheduling.

La prelazione deve avvenire anche quando un processo con maggior priorità diventa ready.

Eventi: utilizzo risorse

Si trattano di eventi che si presentano quando un processo richiede una risorsa che non può essere allocata subito, e il gestore dell'evento ne cambia lo stato in *blocked*.

Se un processo rilascia una risorsa con una System call, il gestore non deve cambiare lo stato di tale processo; invece, deve controllare se altri processi sono in attesa di tale risorsa ed allocarla eventualmente ad uno di essi cambianone lo stato da *blocked* a *ready*.

Azioni di gestione simili avvengono anche quando vengono invocate System call per l'avvio, l'interrupt o il completamento di **operazioni di I/O**.

GESTIONE DEGLI EVENTI: ECB

Quando si verifica un evento, il kernel deve trovare il processo il cui stato è affetto dalla sua concorrenza. Per velocizzare il tutto, i SO usano vari schemi come ad esempio l'**Event Control Block** (ECB).

Il kernel si costruisce una lista di eventi associata ad ogni categoria di risorse così da ridurre i tempi, così che quando si verifica una interrupt si cerca l'evento direttamente nella lista degli ECB.

Individuato l'evento, viene estratto del processo affetto da quell'evento, e il suo stato è modificato in *ready*.

Event description
Process id
ECB pointer

CONDIVISIONE, COMUNICAZIONE e SINCRONIZZAZIONE TRA PROCESSI

Tra processi possono avvenire quattro tipi di interazione:

Tipo di interazione	Descrizione
Condivisione dei dati	I dati condivisi possono diventare inconsistenti se diversi processi li modificano allo stesso tempo. Per questo motivo i processi devono interagire per decidere quando è possibile per un processo modificare o usare i dati condivisi in modo sicuro.
Scambio di messaggi	I processi scambiano informazioni inviando messaggi l'uno all'altro.
Sincronizzazione	Per ottenere un obiettivo comune, i processi devono coordinare le proprie attività.
Segnali	Un segnale è usato per comunicare a un processo l'occorrenza di una situazione di eccezione.

SEGNALI

Un segnale è usato per notificare una situazione eccezionale ad un processo e permettergli di occuparsene immediatamente.

Tali situazioni e i nomi/numeri dei segnali sono definiti nel SO. Alcuni sono associati all'hardware (come una condizione di overflow della CPU), altri sono associati ai processi figli, altri all'utilizzo di risorse, e altri ancora alle comunicazioni di emergenza da un utente ad un processo (terminazione da parte dell'utente).

Tali segnali possono essere *sincroni* o *asincroni*, e sono gestiti dal *signal handler* o da *handler di default*.

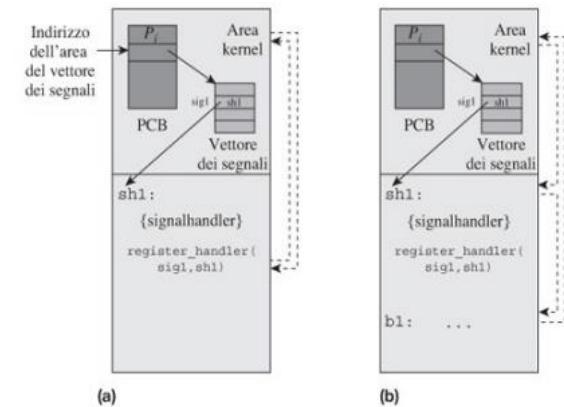
I segnali per essere gestiti dal SO in vari modi, ma possiamo distinguere **due situazioni** principali:

Tipicamente ogni SO ha una System call al quale passiamo il segnale del quale vogliamo gestire l'occorrenza e la funzione da eseguire.

Il primo caso (a) consiste che nel momento stesso in cui viene ricevuto il segnale viene richiamata la System call. In questo momento il kernel richiama il processo dal PCB a cui è associato il segnale, questo smette di fare qualsiasi cosa stesse facendo, e si occupa di gestire il segnale saltando all'indirizzo dell'handler del segnale. Infine, si ritorna al programma principale.

Il secondo caso (b) consiste che prima che venisse eseguita una certa istruzione con indirizzo b1, il processo è stato prelazionato e poi viene ricevuto il segnale.

In tal caso l'esecuzione del processo è dirottata al signal handler ed è ripristinata dopo l'esecuzione dello stesso.



LEZIONE 5 – THREAD

Un thread è l'esecuzione di un programma che usa le risorse di un processo.

Si tratta di un modello alternativo di esecuzione di un programma, il quale viene creato da un processo attraverso una System call e lavora all'interno del contesto del processo stesso.

L'uso di thread suddivide di fatto lo stato di un processo in due parti:

- lo stato della risorsa resta con il processo;
- lo stato della CPU è associato con il thread.

Lo **scopo** di un thread è quello di **ridurre** l'overhead generato dai processi.

Ricordiamo che le applicazioni usano i processi concorrenti per velocizzare la loro esecuzione. Tuttavia, la commutazione tra i processi genera un elevato overhead dovuto alla quantità di informazione da salvare e caricare ad ogni commutazione.

Ricordiamo che possiamo dividere un processo in due parti: *contesto del processo* e *PCB*, ovvero nella parte che **"usa le risorse"** e quella che riguarda l' **"esecuzione"**:

- Per quanto riguarda il PCB non possiamo fare nulla, in quanto sono operazioni necessarie e che non possono essere evitate. Possiamo considerarlo un overhead "fisso".
- Per quanto riguarda il contesto si può invece agire sull'utilizzo delle risorse.

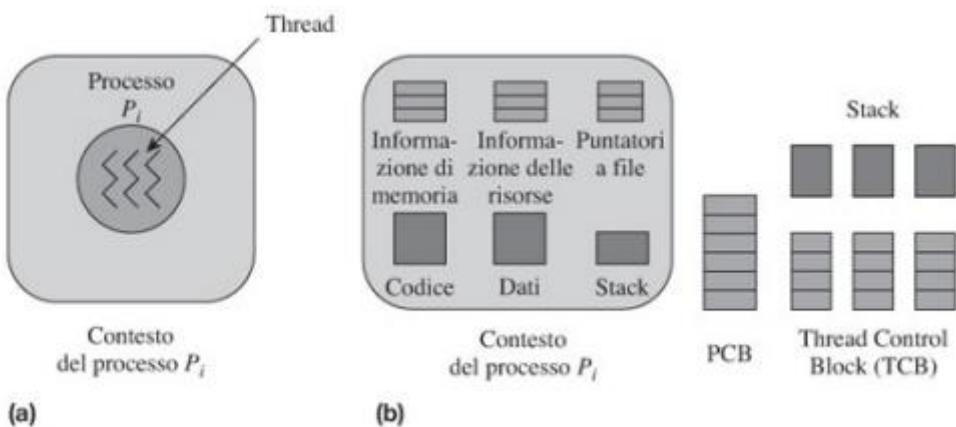
Infatti posso limitare la commutazione solo all'esecuzione, e non fare alcuna commutazione a livello di risorse, rendendo l'overhead più contenuto.

Il problema che si viene a creare quando un processo genera altri processi figli, è che tali figli ereditano il contesto del padre, ovvero è come se ne fossero delle copie (ovviamente, le informazioni di esecuzione sono diverse). Si crea quindi ridondanza ed overhead inutile.

I thread nascono proprio per sopperire a questo problema, in quanto le informazioni di stato (ovvero il contesto) sono condivise fra tutti i thread.

Ciò significa che quando dovrò commutare da un thread all'altro dovrò solo salvare le informazioni relative all'esecuzione, generando **molto meno overhead**.

Visualizzazione di un thread



(a) rappresenta concettualmente un processo con thread. Notiamo come tutti i thread condividano lo stesso contesto.

(b) rappresenta l'implementazione di un processo con thread, il quale è caratterizzato da un contesto e dal PCB. Quando vengono creati dei thread avviene qualcosa di simile ai processi, in quanto vengono creati dei Thread Control Block (TCB) che contengono informazioni legate all'esecuzione, ed ogni thread ha un suo stack.
Ovviamente tutti i thread di un processo condividono il contesto di quel processo.

TCB

Il Thread Control Block contiene le informazioni per la schedulazione dei thread, quindi *thread id*, *priorità* e *stato*.

Ogni thread ha un proprio stato della CPU: PSW e GPR.

Ogni thread ha un puntatore al PCB del processo all'interno del quale è stato creato, ed un puntatore al TCB per creare la lista necessaria alla schedulazione.

TRANSIZIONI DI STATO

I thread hanno stato e transizioni di stato dei processi.

Appena creati essi sono impostati allo stato *ready*, in quanto al momento di creazione hanno già le risorse necessarie.

Esso transisce nello stato *running* quando è disposto a dispatching, ed entra nello *blocked* a causa dei requisiti di sincronizzazione del processo.

Notare che un thread non può andare di per sé nello stato *blocked* in quanto *non* può fare richieste di risorse, ma è il processo che ne fa richiesta.

Inoltre, i thread non sono soggetti a swap-in o swap-out, ma lo swap può avvenire solo sul processo.

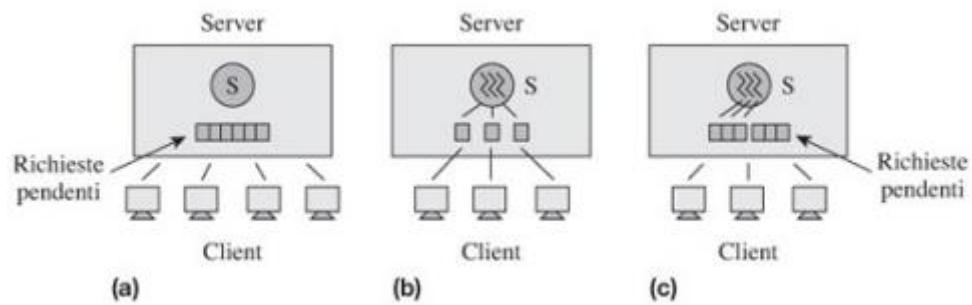
VANTAGGI THREAD vs PROCESSI

Vantaggio	Motivazione
Minore overhead dovuto alla creazione e alla commutazione	Lo stato del thread consiste solo dello stato dell'elaborazione. Lo stato dell'allocazione delle risorse e lo stato delle comunicazioni non sono parte dello stato del thread, per cui la creazione e la commutazione dei thread produce un overhead inferiore.
Comunicazione più efficiente	I thread di un processo possono comunicare tra loro attraverso dati condivisi, evitando in questo modo l'overhead di comunicazione dovuto alle chiamate di sistema.
Progettazione semplificata	L'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente.

Esempio di utilizzo di processi e thread

Per la gestione di un insieme di richieste da lato client a server, posso adottare varie strategie:

- (a) server che usa **codice sequenziale**. Accoda tutte le richieste dei client;
- (b) server **multi-thread**. Per ogni richiesta crea un thread, che vengono eseguite in modo concorrente. L'overhead è ridotto e quindi più efficiente.
- (c) server che usa un **pool di thread**; Crea all'inizio un insieme di thread e li riutilizza. Anche in questo le richieste vengono eseguite in modo concorrente, solo che i thread non vengono creati (se ce ne sono abbastanza) e distrutti ogni volta ma riutilizzati. I thread vengono terminati solo alla fine del processo.



CODIFICA PER USARE I THREAD

Quando utilizzo i thread devo accertarmi della correttezza della condivisione dei dati. Per tal motivo utilizziamo delle librerie **thread safe**.

Inoltre la **gestione dei segnali** si complica con l'uso dei thread, in quanto ci si chiede quale thread deve gestire il segnale.

Tale scelta può essere effettuata dal kernel o dall'applicazione, ma normalmente se il segnale è connesso ad un'eccezione hardware allora tale segnale è gestito dal thread che lo ha generato, mentre in tutti gli altri casi può essere gestito da uno qualunque dei thread del processo (idealmente è quello a priorità maggiore).

Thread in C

I thread in C sono definiti dallo standard Posix, la quale definisce la libreria **pthread** che mette a disposizione 60 routine per le gestione dei thread e per fornire assistenza per la condivisione dei dati e per la sincronizzazione.

THREAD DI LIVELLO KERNEL, UTENTE e IBRIDI

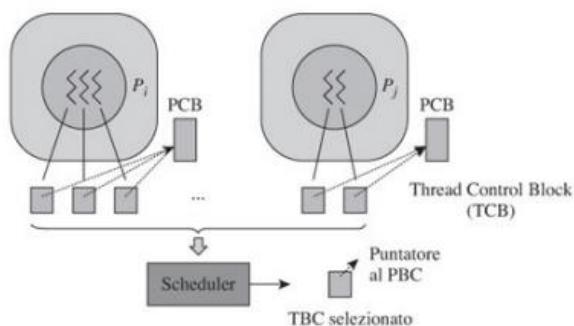
Diverse tipologie di thread impattano le prestazioni in diversi modi, ed ognuno ha dei vantaggi e degli svantaggi. Distinguiamo 3 tipi di thread:

- **thread di livello kernel**: gestiti dal kernel;
- **thread di livello utente**: gestiti dalla libreria dei thread;
- **thread ibridi**: combinazione dei thread di livello kernel e utente.

Livello Kernel

Un thread di livello kernel è come un processo ma con una minore quantità di informazioni di stato. Ovviamente essendo coinvolto il kernel per la commutazione tra thread si genera un certo overhead in quanto viene eseguito codice kernel.

Un thread può essere creato attraverso una System call, e ad ogni thread verrà associato un TCB. Lo scheduler selezionerà poi un certo thread attraverso il quale potrà risalire al processo che lo ha generato (in quanto nel thread è presente il puntatore al PCB del suo processo).



Quando si presenta un **evento**, il kernel salva lo stato della CPU del thread interrotto nel suo TCB, gestisce l'evento, e poi lo scheduler seleziona un thread *ready* dalla lista dei TCB.

Il dispatcher poi verifica se tale thread appartiene ad un processo diverso rispetto a quello del thread interrotto: in caso affermativo vengono caricate sia le informazioni del contesto che dell'esecuzione, altimenti vengono caricate solo le informazioni dell'esecuzione.

Vantaggi

- I Thread sono simili ai processi (con meno informazioni), e quindi sono convenienti per i programmatori.
- In ambienti multi CPU consentono il parallelismo.

Svantaggi

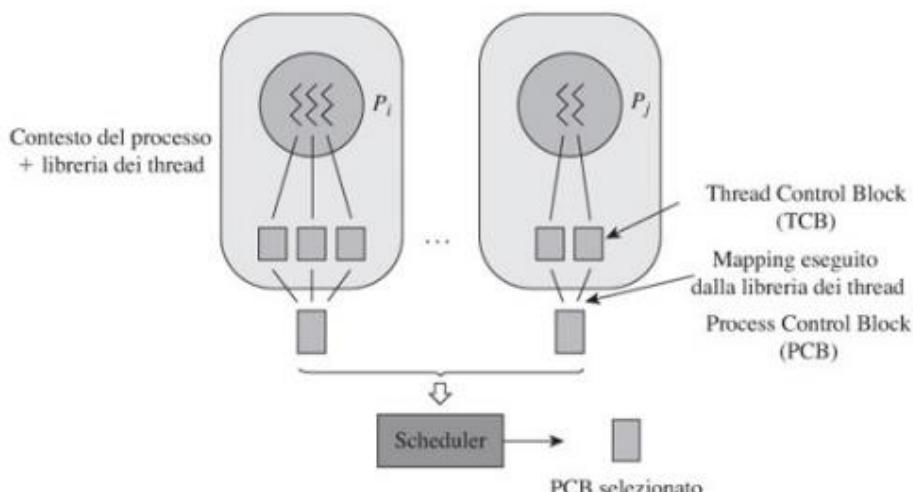
- Quando è gestito un evento, la commutazione tra thread è fatta dal kernel, e quindi tolgo tempo di esecuzione ai programmi utente generando overhead.

Tuttavia, nonostante il risparmio di overhead sia limitato, risulta comunque essere un modello più conveniente rispetto ad uno che non implementa thread.

Thread di livello Utente

Un thread di livello utente viene gestito direttamente dalla libreria ed ha una commutazione dei thread più veloce rispetto a quelli di livello kernel in quanto il kernel stesso non è coinvolto.

Sarà compito della libreria occuparsi della corrispondenza fra i thread di livello utente con i PCB associati al processo.



La libreria mantiene lo *stato* del thread esegue la commutazione. La creazione di nuovo thread avviene attraverso i metodi della libreria.

Vantaggi

- La schedulazione e la sincronizzazione è gestita dalla libreria, evitando l'utilizzo di System call. Ergo, la commutazione tra thread è meno oneroso rispetto ai thread di livello kernel.
Inoltre da anche la possibilità ad un processo di scegliere la strategia di scheduling più adatta alla sua natura.

Svantaggi

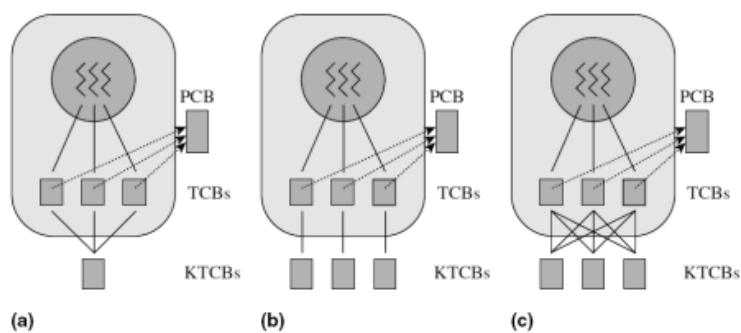
- Il kernel non sa della distinzione tra thread e processo. Per tal motivo un thread bloccato su una System call fa sì che il kernel blocchi l'intero processo.
- Può essere operativo un solo thread per volta. Per tal motivo i thread di livello utente non forniscono **parallelismo** e la **concorrenza** viene seriamente compromessa se si verifica un blocco.

Thread Ibridi

Un modello di Thread Ibridi è una combinazione di **parallelismo** e **basso overhead**, che erano le caratteristiche che mancavano rispettivamente nei thread di livello utente e quelli di livello kernel.

Per la precisione, possiamo distinguere 3 modelli:

- a) **molti a uno**: a più thread di livello utente è associato un thread di livello kernel.
Con tale modello ho quanto meno overhead possibile.
- b) **uno a uno**: ad ogni thread di livello utente è associato un thread di livello kernel.
Con tale modello posso sfruttare al massimo il parallelismo. (es Unix, Linux)
- c) **molti a molti**: a più thread di livello utente posso associare più thread di livello kernel. Con tale modello ho un mix equilibrato fra parallelismo e basso overhead, tuttavia è molto complicato da gestire ed è stato via via abbandonato dai vari SO.
(es. Solaris fino alla versione 8).



LEZIONE 6 – SINCRONIZZAZIONE DEI PROCESSI (1)

La *sincronizzazione dei processi* indica le tecniche usate per ritardare e ripristinare i processi per implementare le interazione tra i processi.

I processi che non interagiscono sono detti *processi indipendenti*.

Notare inoltre che, in questo caso, il termine processo è un termine generico usato sia per processi che per thread.

Formalmente, definite le operazioni `read_seti` e `write_seti`, dove:

- Con `read_seti` si intende l'insieme dei dati letti dal processo P_i e messaggi interprocesso o segnali ricevuti da P_i ;
- Con `write_seti` si intende l'insieme dei dati modificati dal processo P_i e messaggi interprocessi o segnali inviati da P_i ;

diremo che:

I processi P_i e P_j sono **processi interagenti** se la `write_set` di uno dei processi si sovrappone con la `write_set` o la `read_set` dell'altro.

$$(read_set_i \cup write_set_i) \cap (read_set_j \cup write_set_j) \neq \emptyset$$

Per convenzione, si utilizza il costrutto che va da **Parbegin** a **Parend** per indicare i processi che vengono eseguiti in concorrenza/parallelismo.

Parbegin	S_{11}	S_{21}	\dots	S_{n1}
	\vdots	\vdots		\vdots
	S_{1m}	S_{2m}	\dots	S_{nm}
Parend	<u>Process P_1</u>		<u>Process P_2</u>	<u>Process P_n</u>

dove le istruzioni $S_{11} \dots S_{1m}$ rappresentano le istruzioni del processo P_1 .

- Le variabili condivise vengono dichiarate prima di un **Parbegin**.
- Le variabili locali vengono dichiarate all'inizio di un processo.
- I commenti sono racchiusi da “{ }”.
- L'indentazione è utilizzata per mostrare le strutture di controllo.

RACE CONDITION

Gli accessi non coordinati ai dati condivisi possono compromettere la consistenza dei dati. Quando si ha uno stato *non sicuro* del valore di una risorsa condivisa si parla di **race condition** su quella risorsa.

Osservazione:

Consideriamo due processi P_i e P_j che aggiornano il valore di d_s con, rispettivamente, le operazioni a_i e a_j .

Indichiamo con $f_i(d_s)$ il risultato dell'operazione a_i , e con $f_j(d_s)$ il risultato dell'operazione a_j .

Queste due operazioni vengono svolte in modo concorrente, tuttavia non sappiamo quale due verrà svolta prima, e di conseguenza non saremo in grado di prevedere il valore di d_s . (ovviamente, se non prendiamo precauzioni in termini di sincronizzazione).

Da tale osservazione, **formalmente** dirò che:

La **race condition** è una condizione in cui il valore di un oggetto condiviso d_s , risultante dall'esecuzione delle operazioni a_i e a_j su d_s nei processi interagenti, può essere diversa da ambo $f_i(f_j(d_s))$ e $f_j(f_i(d_s))$.

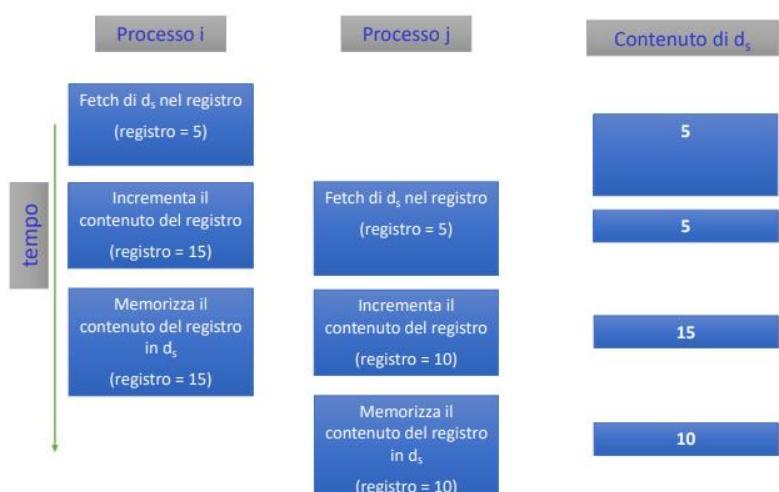
Ovvero, si ha *race condition* quando il risultato è diverso sia da quello dato dal processo P_j che modifica un certo oggetto condiviso che viene poi modificato da P_i , che da quello dal processo P_i che modifica lo stesso oggetto condiviso che viene poi modificato da P_j .

Un esempio in cui si può verificare race condition è quando due processi stanno cercando di incrementare il valore della stessa variabile: come sappiamo il processo deve prima prelevare il valore dal registro (fetch), incrementarlo e poi memorizzarlo;

Può capitare che le due operazioni si sovrappongano, e nel mentre un processo sta incrementando quel valore contenuto nel registro, un altro processo faccia il fetch in quel registro. Essendo che il primo processo non ha ancora memorizzato quel dato, il secondo processo avrà prelevato un valore non aggiornato che andrà poi ad incrementare a sua volta.

Quello che accadrà è che il primo processo memorizzerà infine il dato su cui avrà fatto il suo incremento, il quale però verrà sovrascritto dal secondo processo il quale memorizzerà il dato su cui avrà fatto il suo di incremento.

Il valore finale dunque sono sarà quello atteso.



Le race condition sono **prevenute** attraverso la **mutua esclusione**, ovvero consentiamo ad un'una sola operazione per volta di accedere ad un dato condiviso.

Quando viene assicurata la mutua esclusione, possiamo essere sicuri che il risultato delle operazioni a_i e a_j sarà o $f_i(f_j(d_s))$ oppure $f_j(f_i(d_s))$.

A questo punto, è fondamentale capire se un dato condiviso è soggetto a race condition. Per tal motivo dobbiamo introdurre un nuovo insieme *update_set*:

- con **update_set_i** si intende l'insieme di dati aggiornati dal processo P_i (letti, modificati, scritti).

Per prevenire una race condition dobbiamo assicurarci che non ci sia *intersezione* fra le operazioni di aggiornamento:

$$update_set_i \cap update_set_j \neq \emptyset$$

Se l'intersezione non è vuota, individuo i dati condivisi e quei dati dovranno essere accessi in mutua esclusione.

SEZIONI CRITICHE

La mutua esclusione è implementata usando le *sezioni critiche* di codice, ove:

una **sezione critica**, per un oggetto d_s , è una sezione di codice che è progettata in modo che non possa essere eseguita concorrentemente con se stessa o con altre sezioni critiche per d_s .

Se un processo P_i sta eseguendo una SC per d_s , un altro processo che intende eseguire una SC per lo stesso dato dovrebbe attendere l'esecuzione di P_i per quella SC.

esempio: per convenzione indichiamo le SC con rettangoli grigi

<pre>if nextseatno ≤ capacity then allottedno:=nextseatno; nextseatno:=nextseatno+1; else display "sorry, no seats available";</pre>	<pre>if nextseatno ≤ capacity then allottedno:=nextseatno; nextseatno:=nextseatno+1; else display "sorry, no seats available";</pre>
<u>Process P_i</u>	<u>Process P_j</u>

Se il processo P_i entra in sezione critica prima di P_j , il processo P_j dovrà attendere finché P_i non sarà uscito dalla sua SC.

Ovviamente, l'utilizzo di SC causa **ritardi** nelle operazioni dei processi ed è compito del processo e del kernel **minimizzare** questi ritardi:

- un processo non deve essere eseguito a lungo in una SC.
- Il processo non deve invocare chiamate di sistema che possono portarlo in uno stato bloccato, all'interno di una SC.
- Il kernel non deve prelazionare un processo che è alle prese con l'esecuzione di una SC.

Per quanto riguarda l'ultimo punto, il kernel deve essere sempre informato se un processo è in una SC, cosa che non può avvenire se si tratta di un processo utente (a meno che non si utilizzino particolari accortezze).

(Inoltre, per comodità assumiamo che un processo rimanga in SC per poco tempo).

PROPRIETÀ PER L'IMPLEMENTAZIONE DI UNA SEZIONE CRITICA

Una SC, per essere ben implementata, deve rispettare 3 proprietà fondamentali:

Proprietà	Descrizione
Mutua esclusione	In ogni istante, al più un processo può eseguire una CS per un dato d_s .
Progresso	Quando nessun processo sta eseguendo una CS per un dato d_s , sarà concesso l'accesso a uno dei processi che vogliono entrare in una CS per d_s .
Attesa limitata	Dopo che un processo P_i ha manifestato il suo interesse a entrare in una CS per d_s , il numero di volte che gli altri processi possono ottenere l'accesso a una CS per d_s prima di P_i è limitato da un intero finito.

Il *progresso* e l'*attesa limitata* insieme prevengono la **starvation**, la quale è una situazione in cui un certo processo cerca di ottenere una risorsa (o in questo caso cerca di entrare in una SC) ma viene sempre scavalcato da qualcun altro processo.

SINCRONIZZAZIONE DI CONTROLLO e OPERAZIONI INDIVISIBILI

Si parla di **sincronizzazione di controllo** ogni qualvolta vogliamo assicurarsi che un certo processo esegua una certa operazione solo dopo che è stata eseguita un'altra operazione da un altro processo.

Una tecnica generale per la sincronizzazione di controllo è la **segnalazione**.

Un **tentativo ingenuo** di segnalazione potrebbe essere fatto mediante **variabili booleane**, che utilizziamo per segnalare quando un certo processo P_i è bloccato e quando una certa operazione a_j è stata eseguita.

Se P_i ad un certo punto ha bisogno di sapere se a_j è stata eseguita per continuare, fa un controllo sulla corrispondente variabile booleana e se è falso si mette in attesa che il processo P_j completi l'operazione a_j .

P_j esegue tranquillamente l'operazione a_j e a fine operazione controlla se P_i è bloccato: in tal caso lo "sblocca" altrimenti conferma solo che l'operazione a_j è stata completata.

Il **problema** con questa soluzione è che in realtà non funziona, in quanto può capitare che mentre il Processo P_i ha confermato che l'operazione non è stata ancora eseguita, prima che possa bloccarsi, tale processo viene prelazionato in favore di P_j che ancora non sa che P_i è bloccato e quindi non può sbloccarlo in anticipo.

Quello che avverrà è che quando verrà nuovamente schedulato P_i , questo andrà a bloccarsi e rimarrà permanentemente bloccato.

Per tal motivo, si devono invece utilizzare operazioni **atomiche** (o indivisibili).

Il concetto di sincronizzazione di controllo è legato a quello di **operazione invisibile**, la quale è un'operazione su un insieme di oggetti che non può essere eseguita concorrentemente con se stessa o altra operazione su un oggetto incluso nell'insieme.

Un **tentativo migliore** di segnalazione consiste nell'utilizzare le **operazioni indivisibili**, che utilizziamo per segnalare se un'operazione è stata già eseguita.

Consideriamo due operazioni indivisibili $check_{-}a_j$ e $post_{-}a_j$, dove $check_{-}a_j$ è invocata dal processo P_i per verificare se tale a_j è stata già eseguita, mentre $post_{-}a_j$ è invocata dal processo P_j per verificare se P_i è bloccata.

Il ragionamento è lo stesso del caso di prima, ma con la differenza che una volta iniziate tali procedure queste non possono essere interrotte, e quindi non può presentarsi il problema descritto prima.

Le operazioni indivisibili sono effettivamente simili alla mutua esclusione.

caso 1:

```

var
    operation_aj_performed : boolean;
    pi_blocked : boolean;
begin
    operation_aj_performed := false;
    pi_blocked := false;

Parbegin
    . . .
    if operation_aj_performed = false
    then
        pi_blocked := true;
        block (Pi);
        {perform operation ai}
    . . .
    . . .
    . . .
Parend;
end.

```

Process P_i

Process P_j

Un tentativo di segnalazione naïve mediante variabili booleane

Race condition nella sincronizzazione di Processi

Time	Actions of process P _i	Actions of process P _j
t ₁	if action_aj_performed = false	
t ₂		{perform action a _j }
t ₃		if pi_blocked = true
t ₄		action_aj_performed := true
:		
t ₂₀	pi_blocked := true;	
t ₂₁	block (P _i);	

caso 2:

```

procedure check_aj
begin
    if operation_aj_performed=false
    then
        pi_blocked:=true;
        block (Pi)
    end;
procedure post_aj
begin
    if pi_blocked=true
    then
        pi_blocked:=false;
        activate(Pj)
    else
        operation_aj_performed:=true;
    end;

```

Operazioni indivisibili

check_aj e **post_aj** per la segnalazione

APPROCCI ALLA SINCRONIZZAZIONE

Possiamo distinguere 3 tipi di approcci:

- Ciclare vs bloccare;
- Supporto HW per la sincronizzazione dei processi;
- Approcci algoritmici, primitive di sincronizzazione e costrutti di programmazione concorrente.

CICLARE VS BLOCCARE

Se un processo vuole accedere in SC, ma questa è già occupata da un altro processo, allora il primo processo dovrà *aspettare* o *bloccarsi*.

Un modo di **aspettare** è attraverso l'**attesa attiva** (busy wait): il processo cicla a vuoto all'interno di un while finché un altro processo è in SC oppure sta venendo eseguita un'operazione indivisibile.

Terminata l'attesa, il processo entra in sezione critica.

Tale metodo è funzionale ma ha molte **conseguenze**:

- non può fornire la proprietà di attesa limitata;
- degrado delle prestazioni a causa del ciclare;
- deadlock.

Per evitare le attese attive, un processo in attesa di entrare in una SC è posto in uno stato **bloccato**, il quale viene poi cambiato in *pronto* solo quando può entrare nella SC: il processo controlla (una sola volta) se un altro processo è in SC oppure sta venendo eseguita un'operazione indivisibile.

In tal caso, il processo fa una System Call per bloccare se stesso, in attesa che qualcun altro prima o poi lo sblocchi (es. kernel).

```
while (some process is in a critical section on {ds} or  
      is executing an indivisible operation using {ds})  
{ do nothing }
```

Critical section or
indivisible operation
using {d_s}

```
if (some process is in a critical section on {ds} or  
      is executing an indivisible operation using {ds})  
then make a system call to block itself;
```

Critical section or
indivisible operation
using {d_s}

A questo punto il processo deve decidere se ciclare o bloccarsi, ma anche tale decisione è soggetta a race condition. Per poter eseguire in modo sicuro questo tipo di operazioni dobbiamo utilizzare la mutua esclusione, e possiamo usare un approccio algoritmo o usare istruzioni della macchina a livello HW.

SUPPORTO HW PER LA SINCRONIZZAZIONE DEI PROCESSI

L'HW mi può supportare mettendomi a disposizione due primitive (fra le altre), le quali sono *indivisibili* in quanto fornite direttamente dall'HW, e che mi permettono di evitare la race condition sulle locazioni di memoria e di implementare dunque la mutua esclusione.

Tali istruzioni sono usate con una **variabile di lock**, necessaria per implementare la SC e le operazioni indivisibili. Si tratta di una variabile che può avere due stati: *aperto* o *chiuso*. Se il lock è aperto possiamo andare in SC e chiudiamo il lock. Usciti dalla SC lo riapriamo. Se il lock è chiuso significa che qualche altro processo è in SC e quindi cicliamo di nuovo sperando il lock sia stato aperto.

Per evitare race condition nell'impostazione della variabile di lock, viene utilizzata un'operazione indivisibile per la lettura e la chiusura, che chiamiamo *entry test*.

Per **implementare** le sezioni critiche e le operazioni indivisibili possiamo utilizzare due istruzioni indivisibili equivalenti: *test-and-set* e *swap*.

- **test-and-set**: pone il valore del lock a 1 ,cioè “chiuso”, e ritorna il valore precedente. Entrambi le istruzioni sono eseguite in modo atomico.
Se il valore restituito era 1, cioè il lock era chiuso, richiamiamo entry test, altrimenti procedo nella SC ed una volta uscito imposterò il lock a 0, cioè aperto.
- **swap**: non opera direttamente sul valore di lock ma utilizza una variabile temporanea *temp* che inizializzerà ad 1. Poi farà uno scambio fra il valore di lock e il valore di *temp*. *temp* adesso contiene il vecchio valore di lock, e verifichiamo se tale valore è aperto.
Se il valore restituito era 1, cioè il lock era chiuso, richiamiamo entry test, altrimenti procedo nella SC ed una volta uscito imposterò il lock a 0, cioè aperto.

TestAndSet	LOCK	DC X'00'	Lock è inizializzato a aperto
	ENTRY_TEST	TS LOCK	Test-and-set lock
		BC 7, ENTRY_TEST	Cicla se lock è chiuso
		...	{Sezione critica o operazione indivisibile}
		MVI LOCK, X'00'	Apre il lock(spostando gli 0)

SWAP	TEMP	DS 1	Riserva un byte per TEMP
	LOCK	DC X'00'	Lock è inizializzato a aperto
		MVI TEMP, X'FF'	X'FF' è utilizzato per chiudere il lock
	ENTRY_TEST	SWAP LOCK, TEMP	
		COMP TEMP, X'00'	Esamina il vecchio valore di lock
		BC 7, ENTRY_TEST	Cicla se lock è chiuso
		...	{Sezione critica o operazione indivisibile}
		MVI LOCK, X'00'	Apre il lock

APPROCCI ALGORITMICI, PRIMITIVE DI SINCRONIZZAZIONE e COSTRUTTI DI PROGRAMMAZIONE CONCORRENTE

Per realizzare le operazioni di sincronizzazione abbiamo varie possibilità, le quali si sono sviluppate nel tempo con lo scopo di risolvere le falte della soluzione precedente:

- **Approcci algoritmici:** sono utilizzati per implementare la mutua esclusione essendo allo stesso tempo indipendenti dalla piattaforma HW o SW. Tuttavia sono complessi e difficili da dimostrare.
- **Primitive di sincronizzazione:** sono implementate usando istruzioni indivisibili e con il supporto del kernel, e sono rappresentate, ad esempio, dalle istruzioni *wait* e *signal* dei semafori. Tuttavia il loro uso improprio può generare molti problemi.
- **Costrutti di programmazione concorrente:** sono utilizzate per evitare l'uso improprio delle operazioni di sincronizzazione, e un esempio ne sono i *monitor*.

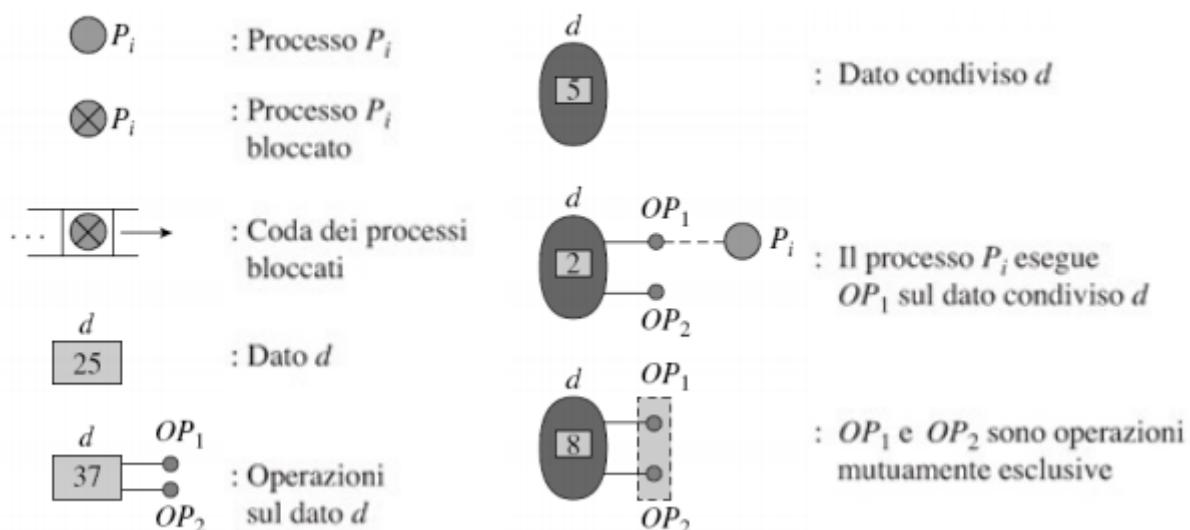
STRUTTURA DEI SISTEMI CONCORRENTI

Nei sistemi concorrenti possiamo distinguere **tre componenti chiave**:

- **Dati condivisi**, cioè i dati dell'applicazione usati e manipolati dai processi e i dati di sincronizzazione;
- **Operazioni sui dati condivisi**;
- **Processi interagenti**.

Una vista del sistema in un istante specifico prende il nome di *istantanea* (snapshot).

Convenzioni grafiche per le istantanee dei sistemi concorrenti



LEZIONE 7 – SINCRONIZZAZIONE DEI PROCESSI (2)

Una soluzione ad un processo di sincronizzazione dovrebbe soddisfare tre **criteri** importanti:

- **correttezza**: le operazioni devono essere fatte in mutua esclusione;
- **massima concorrenza**: devo far concorrere quanti più processi possibili;
- **nessuna attesa attiva**: sostanzialmente, devo evitare il busy waiting.

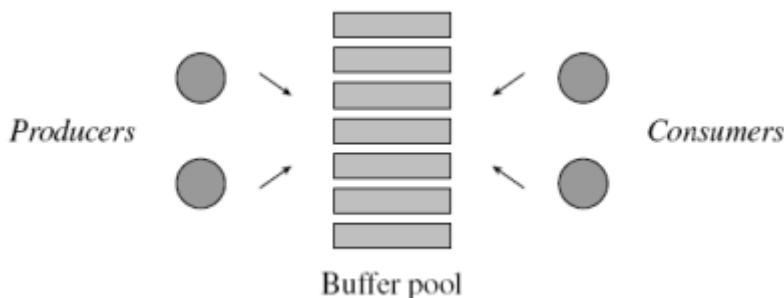
La maggior parte dei problemi di sincronizzazione si rifanno alle soluzioni di alcuni problemi *classici* di sincronizzazione, quali: Produttori-Consumatori con buffer limitati; Lettori e scrittori; Filosofi a cena.

Produttori-Consumatori con buffer limitati

Si tratta di un problema in cui si ha un insieme di buffer, un certo numero di produttori e un certo numero di consumatori.

I produttori sono i processi che memorizzano delle informazioni in questi buffer, mentre i consumatori sono i processi che leggono (*consumano*) da questi buffer.

Il **problema** è che tutto l'insieme dei buffer è soggetto a *race condition*, e per tal motivo dobbiamo sincronizzare le operazioni fra i produttori e i consumatori in modo che le operazioni si svolgano correttamente.



Una **soluzione** deve soddisfare:

1. Un produttore non deve sovrascrivere un buffer pieno.
2. Un consumatore non deve consumare un buffer vuoto.
3. Produttori e consumatori devono accedere ai buffer in modo mutuamente esclusivo.
4. (opzionale) Le informazioni devono essere consumate nello stesso ordine in cui sono messe nel buffer.

Una **prima soluzione** può consistere nell'utilizzare delle *variabili booleane* per verificare se un produttore ha *prodotto* o se un consumatore ha *consumato*.

Se un produttore non sta producendo nulla, allora verifica se esiste un buffer vuoto, e in tal caso produce in tale buffer.

Simmetricamente, se un consumatore non sta consumando nulla, allora verifica se esiste un buffer pieno, e in tale consuma da tale buffer.

Ovviamente le operazioni di ricerca del buffer e di successiva produzione/consumazione devono avvenire in mutua esclusione.

Tale soluzione soffre di **due problemi**: c'è poca concorrenza e sono presenti *attese attive*.

Per **migliorare** tale schema dobbiamo mantenere la mutua esclusione per l'accesso ai buffer, però se un produttore inserisce un nuovo elemento nel buffer dovrà segnalarlo al consumatore e, viceversa, se un consumatore estrae un elemento dal buffer dovrà segnalarlo al produttore.

Insomma, si tratta di aggiungere una **segnalazione**.

Supponiamo di avere solo un produttore, consumatore e buffer, avremo il seguente **schema**:

```
var
    buffer : . . . ;
    buffer_full : boolean;
    produttore_bloccato, consumatore_bloccato : boolean;
begin
    buffer_full := false;
    produttore_bloccato := false;
    consumatore_bloccato := false;
Parbegin
repeat
    check_b_empty;
    {Inserisci nel buffer}
    post_b_full;
    {Parte restante del ciclo}
all'infinito;
Parend;
end.
```

Produttore Consumatore

Il **produttore** verificherà se il buffer è vuoto attraverso l'operazione *check_b_empty* e in caso affermativo effettuerà la produzione. Poi segnalerà al consumatore che tale buffer è pieno attraverso l'operazione *post_b_full*.

Nel caso il buffer fosse pieno, allora il produttore si arresterà in attesa che il buffer diventi vuoto.

Viceversa, il **consumatore** verificherà se il buffer è pieno attraverso l'operazione *check_b_full* e in caso affermativo effettuerà la consumazione. Poi segnalerà al produttore che tale buffer è vuoto attraverso l'operazione *post_b_empty*.

Nel caso il buffer fosse vuoto, allora il consumatore si arresterà in attesa che il buffer diventi pieno.

codice delle operazioni:

```
procedure check_b_empty
begin
    if buffer_full = true
    then
        produttore_bloccato := true;
        block (produttore);
    end;

procedure post_b_full
begin
    buffer_full := true;
    if consumatore_bloccato = true
    then
        consumatore_bloccato := false;
        activate (consumatore);
    end;
```

Operazioni del produttore

```
procedure check_b_full
begin
    if buffer_full = false
    then
        consumatore_bloccato := true;
        block (consumatore);
    end;

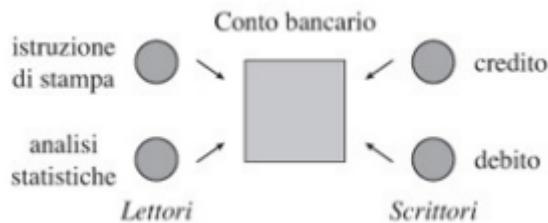
procedure post_b_empty
begin
    buffer_full := false;
    if produttore_bloccato = true
    then
        produttore_bloccato := false;
        activate (produttore);
    end;
```

Operazioni del consumatore

Lettori e Scrittori

Si tratta di un problema in cui si ha un dato condiviso, un certo numero di lettori e un certo numero di scrittori.

I lettori sono i processi che leggono il dato, mentre gli scrittori sono i processi che sovrascrivono il dato.



Una **soluzione** deve soddisfare:

1. Molti lettori possono leggere contemporaneamente.
2. La lettura è proibita mentre uno scrittore sta scrivendo.
3. Solo uno scrittore può eseguire la scrittura in un dato momento.
4. (opzionale) Diamo preferenza ai lettori o ai scrittori.

Un *lettore* innanzitutto deve verificare se uno scrittore sta scrivendo: in caso affermativo aspetta, altrimenti può direttamente leggere (non ci interessa se qualcun altro sta già leggendo in quanto i lettori possono essere in concorrenza).

Una volta letto, il lettore verifica se qualcun altro sta leggendo: se nessuno sta leggendo allora attiva uno scrittore.

Uno *scrittore* innanzitutto deve verificare se un lettore sta leggendo o se un altro scrittore sta scrivendo: in caso affermativo aspetta, altrimenti potrà scrivere (ed è un'operazione che deve avvenire in mutua esclusione).

Una volta scritto, verifica se ci sono lettori o scrittori in attesa ed in caso li attiva (questa parte dipende da chi abbiamo dato precedenza).

Parbegin repeat If uno scrittore sta scrivendo then { attendi }; { leggi } If nessun lettore sta leggendo then if uno scrittore/i è in attesa then attiva uno scrittore in attesa; forever; Parend; end.	repeat If un lettore/i sta leggendo, o uno scrittore sta scrivendo then { attendi }; { scrivi } If un lettore/i o scrittore/i sono in attesa then attiva uno scrittore o un lettore in attesa; forever;
---	---

Lettore/i

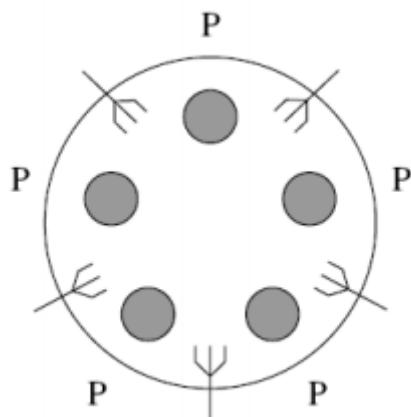
Scrittore/i

Filosofi a cena

Si tratta di un problema in cui si ha una tavola rotonda ed un insieme di filosofi, e dove ogni filosofo deve mangiare ogni qualvolta voglia mangiare prima che muoia di inedia.

Un filosofo, per mangiare, deve utilizzare sia la forchetta alla sua sinistra che quella a sua destra.

In tal problema è facile che si presentino situazioni di deadlock, quindi tutti i processi sono bloccati, ed inoltre possono anche presentarsi situazioni di livelock, ovvero ci sono processi che favoriscono sempre gli altri.



Una **prima soluzione** consiste nel richiedere che il filosofo prelevi una forchetta oer volta (es. prima sx e poi dx). Se la prima forchetta non è disponibile si blocca, altrimenti procede a prendere la seconda forchetta, e se questa non è disponibile si blocca.

Se il filosofo ha entrambi le forchette può mangiare.

Finito di mangiare, il filosofo posa entrambe le forchette e poi controllo se i suoi vicini ne hanno bisogno: se il vicino di sinistra ha bisogno della forchetta di destra, attiva tale vicino; se il vicino di destra ha bisogno della forchetta di sinistra, attiva anche quest'altro vicino. Poi si rimette a pensare.

Il **problema** di questa soluzione è proprio la presenza di potenziali deadlock o race condition, a meno che: se presa la prima forchetta quell'altra non è disponibile, rilascia la prima forchetta e riprova più tardi; questa soluzione soffre tuttavia di livelock.

La **soluzione** consiste nel prelevare ambo le forchette contemporaneamente, o più precisamente si tratta di un'azione atomica con cui prima prende una forchetta e poi l'altra (in quanto può prendere una sola forchetta per volta).

Se tale azione riesce, allora il filosofo mangia e poi posa ambo le forchette (op. atomica).

Anche in questo caso, finito di mangiare il filosofo controlla che i suoi vicini hanno bisogno di una forchetta.

Tuttavia anche tale soluzione ha un **problema**, ovvero che tale loop genera un'attesa attiva.

```

var      successful : boolean;
repeat
    successful := false;
    while (not successful)
        if entrambe le forchette sono disponibili then
            prendi le forchette una alla volta;
            successful := true;
        if successful = false
        then
            block ( $P_i$ );
            { mangia }
            posa entrambe le forchette;
        if il vicino di sinistra è in attesa della sua forchetta di destra
        then
            activate (vicino di sinistra);
        if il vicino di destra è in attesa della sua forchetta di sinistra
        then
            activate (vicino di destra);
            { pensa }
    all'infinito

```

APPROCCI ALGORITMICI PER LE SEZIONI CRITICHE

Gli approcci algoritmici per implementare le sezioni critiche non impiegano né il kernel per il blocco e l'attivazione dei processi, né le istruzioni indivisibili HW per evitare le race condition.

Esse sono indipendenti dal SO e dall'HW, tuttavia usano il *busy wait* e sono molto complesse (e difficili da dimostrare).

Distinguiamo in particolar modo:

- algoritmi a *due processi* (fra cui Decker e Peterson)
- algoritmi a *n processi*.

ALGORITMI A DUE PROCESSI

Una **prima soluzione** consiste nello sfruttare una variabile *turn* che detta quale processo deve entrare in sezione critica. Essendo i processi solo 2, turn potrà assumere solo i valori 1 e 2.

P₁ prima di entrare in SC controlla se turn = 2: in tal caso dovrà aspettare (busy waiting). Altrimenti entrerà nella SC ed una volta uscito porrà turn = 2.

La stessa cosa accadrà simmetricamente per P₂.

Tuttavia tale soluzione **viola la condizione del progresso**, in quanto potrebbe capitare che P₁ imposti turn = 2 e si metta ad aspettare, ma nel mentre P₂ non è ancora entrato in sezione critica e quindi P₁ aspetta a vuoto.

Una **seconda soluzione** consiste nello sfruttare, per ogni processo, una variabile di stato c_i che indica se un dato processo P_i è un SC (0 è dentro, 1 è fuori).

P₁ prima di entrare in SC controlla se c₂ = 0, cioè se P₂ è già in SC: in tal caso dovrà aspettare. Altrimenti indicherà che sta per entrare in SC ponendo c₁ = 0, ed una volta uscito porrà c₁ = 1.

La stessa cosa accadrà simmetricamente per P₂.

Tale soluzione non viola la condizione del progresso, tuttavia **viola la condizione di mutua esclusione** quando i processi cercano di entrare quasi simultaneamente, infatti: all'inizio sia c₁ = 0 che c₂ = 0, e può capitare che mentre P₁ effettua il controllo su c₂ e quindi ha l'OK per andare in SC, *prima* che imposti c₁ = 0, esso viene prelazionato in favore di P₂, il quale a sua volta effettua il controllo su c₁ che però non era stato ancora impostato a 0. Di conseguenza anche P₂ entrerà in sezione critica e verrà dunque violata la mutua esclusione.

Per risolvere tale problema potremmo pensare di modificare prima il valore di c₁ o c₂ e poi effettuare il controllo, ma tale soluzione può portare a **deadlock** sempre se i due processi cercano di entrare quasi simultaneamente (entrambi si bloccano sul ciclo while).

Algoritmo 6.1 Primo tentativo

```
var      turn : 1..2;
begin
    turn := 1;
Parbegin
repeat
    while turn = 2
        do {niente};
    {Sezione critica}
    turn := 2;
    {Resto del ciclo}
forever;
Parend;
end.

Processo P1
```

```
repeat
    while turn = 1
        do {niente};
    {Sezione critica}
    turn := 1;
    {Resto del ciclo}
forever;
```

Processo P₂

Algoritmo 6.2 Secondo tentativo

```
var      c1, c2 : 0..1;
begin
    c1 := 1;
    c2 := 1;
Parbegin
repeat
    while c2 = 0
        do {niente};
    c1 := 0;
    {Sezione critica}
    c1 := 1;
    {Resto del ciclo}
forever;
Parend;
end.

Processo P1
```

```
repeat
    while c1 = 0
        do {niente};
    c2 := 0;
    {Sezione critica}
    c2 := 1;
    {Resto del ciclo}
forever;
```

Processo P₂

ALGORITMO DI DEKKER

L'algoritmo di Dekker combina le soluzioni dei primi due algoritmi, quindi fa uso sia della variabile **turn** che delle **variabili di stato**.

Se P_1 e P_2 provano ad entrare contemporaneamente in SC, **turn** indica a quale dei due è consentito.

Se non c'è competizione per entrare, *turn* non ha effetto, tuttavia se entrambi i processi tentano di accedere nelle rispettive SC, *turn* forza una dei due a favorire l'altro.

P_1 innanzitutto indica che vuole entrare in SC ponendo $c_1 = 0$, poi verifica se anche P_2 ha mostrato la propria intenzione di entrare in SC: In tal caso si considera la variabile *turn*.

Se $turn = 2$ significa che è il turno di P_2 , quindi P_1 imposta $c_1 = 1$ e aspetta.

Quando P_2 cambia il valore di *turn* allora P_1 porrà di nuovo $c_1 = 0$ ed entrerà in SC.

Uscito dalla SC, P_1 pone $turn = 2$ (quindi P_2 può entrare di nuovo in SC) e poi pone $c_1 = 1$.

La stessa cosa accadrà simmetricamente per P_2 .

In tutto ciò, *turn* viene preso in considerazione solo se ambo i processi stanno cercando di entrare in SC nello stesso tempo.

Tale algoritmo implementa tutte le **proprietà** delle SC ed inoltre evita deadlock e livelock.

Algoritmo 6.3 Algoritmo di Dekker

```
var   turn : 1..2;
      c1, c2 : 0..1;
begin
  c1 := 1;
  c2 := 1;
  turn := 1;
Parbegin
  repeat
    c1 := 0;
    while c2 = 0 do
      if turn = 2 then
        begin
          c1 := 1;
          while turn = 2
            do { niente };
          c1 := 0;
        end;
      { Sezione critica }
      turn := 2;
      c1 := 1;
      { Resto del ciclo }
    forever;
  Parend;
end.
Processo P1
```

```
repeat
  c2 := 0;
  while c1 = 0 do
    if turn = 1 then
      begin
        c2 := 1;
        while turn = 1
          do { niente };
        c2 := 0;
      end;
    { Sezione critica }
    turn := 1;
    c2 := 1;
    { Resto del ciclo }
  forever;
Processo P2
```

ALGORITMO DI PETERSON

L'idea è simile a quella di Decker ma utilizza un array booleano di *flag*, dove si ha una flag per ogni processo (equivalenti alle variabili di stato c_i).

Anche in questo algoritmo si utilizza **turn**, solo che in questo caso più che indicare quale processo favorire, favorisce *esplicitamente* l'altro processo.

Inoltre cambia anche la **notazione**: si suppone che i due processi siano P_0 e P_1 e gli id (0 e 1) sono usati per accedere ai **flag** di stati,

Se P_1 ha indicato la propria intenzione di entrare in SC e contemporaneamente turn vuole favorire tale processo, allora P_0 aspetterà finché l'altro non sarà uscito dalla sua SC e avrà impostato il suo flag a false.

Vale anche il viceversa.

Tale algoritmo è equivalente a quello di Decker, quindi implementa tutte le **proprietà** delle SC ed inoltre evita deadlock e livelock.

L'unica differenza è che si presenta sotto una forma più *snella*.

Algoritmo 6.4 Algoritmo di Peterson

```
var      flag : array [0..1] of boolean;
         turn : 0..1;
begin
  flag[0] := false;
  flag[1] := false;
Parbegin
  repeat
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1
      do { niente };
      { Sezione critica }
      flag[0] = false;
      { Resto del ciclo }
  forever;
Parend;
end.  
Processo  $P_1$ 
```

```
repeat
  flag[1] := true;
  turn := 0;
  while flag[0] and turn =
    do { niente };
    { Sezione critica }
    flag[1] = false;
    { Resto del ciclo }
  forever;
Processo  $P_2$ 
```

ALGORITMI A N PROCESSI

Sono algoritmi più complessi rispetto a quelli a due processi, ed è necessario conoscere il numero dei processi che entrano in SC. In tal caso, ogni processo controlla lo stato di **altri n-1** processi.

ALGORITMO DEL PANETTIERE

L'idea di tale algoritmo è che ogni processo "prenda" un numero. Il processo che ha il numero più piccolo è **servito**, cioè entra in SC.

Si usano due array di n elementi:

- l'array booleano **choosing**, dove $\text{choosing}[i]$ indica se P_i è impegnato della scelta;
- l'array di interi **number**, dove $\text{number}[i]$ contiene il numero scelto da P_i .
Se $\text{number}[i] = 0$, allora P_i non ha scelto il numero.

Visto che è possibile che due processi abbiano lo stesso numero, allora serviremo il processo che ha la coppia $(\text{number}[i], i)$ minore, dove: nel caso avessero lo stesso numero, daremmo precedenza al processo con indice minore.

All'inizio inizializzeremo tutti gli elementi di *choosing* a false, in quanto nessun processo è ancor impegnato nella scelta, e tutti gli elementi di *number* a 0, in quanto nessun processo ha ancor preso alcun numero.

In **Parbegin**,

il processo P_i dovrà innanzitutto scegliere il numero: imposta $\text{choosing}[i] = \text{true}$, sceglie il numero come *il massimo dei numeri precedenti + 1*, e finito di scegliere imposta $\text{choosing}[i] = \text{false}$;

Il motivo per cui due processi potrebbero avere lo stesso numero, è che mentre P_i sta per effettuare la somma viene prelazionato a favore di un altro processo, il quale potrebbe star effettuando anch'esso la scelta e come numero massimo ottiene lo stesso numero di P_i . Di conseguenza entrambi i processi si ritroveranno con lo stesso numero.

P_i poi aspetta che tutti gli altri processi abbiano preso il loro numero. Poi verifica se qualcun altro processo ha la coppia minore della sua, o se il loro numero è 0: in tal caso aspetterà.

Quando arriverà il turno di P_i esso entrerà in SC, ed una volta uscito imposterà $\text{number}[i] = 0$ per indicare che dovrà scegliere di nuovo il numero.

Tale algoritmo implementa tutte le **proprietà** delle SC ed inoltre evita deadlock e livelock.

NOTARE che l'array *choosing* è fondamentale perché garantisce la mutua esclusione. Infatti senza di esso un processo non aspetterebbe che tutti gli altri prendano il loro numero, il che può portare al nascere di una situazione in cui più processi entrano in SC contemporaneamente.

Algoritmo 6.6 Algoritmo del panettiere (Bakery) (Lamport [1974])

```
const n = ...;
var choosing : array [0..n - 1] of boolean;
    number : array [0..n - 1] of integer;
begin
    for j := 0 to n - 1 do
        choosing[j] := false;
        number[j] := 0;
Parbegin
process Pi :
repeat
    choosing[i] := true;
    number[i] := max (number[0],...,number[n - 1])+1;
    choosing[i] := false;
    for j := 0 to n - 1 do
        begin
            while choosing[j] do { nothing };
            while number[j] ≠ 0 and (number[j],j) < (number[i],i)
                do { nothing };
        end;
        { Sezione critica }
        number[i] := 0;
        { Resto del ciclo }
    forever;
processo Pj : ...
Parend;
end.
```

LEZIONE 8 – SINCRONIZZAZIONE DEI PROCESSI (3)

SEMAFORI

Un semaforo è una **primitiva di sincronizzazione** introdotta per superare i limiti delle soluzioni algoritmiche e permette di implementare facilmente la mutua esclusione.

Si tratta di una variabile condivisa con valori interi non negativa che può essere soggetta alle sole operazioni di:

1. Inizializzazione (specificata come parte della sua dichiarazione);
2. Operazioni indivisibili *wait* e *signal*, atomiche.

La *wait* è un'operazione che *attende* se il valore del semaforo è uguale (o minore) di 0.

In caso contrario decrementa il valore del semaforo di 1.

La *signal* è un'operazione che verifica se qualche processo è bloccato e in caso lo *sveglia*.

In caso contrario incrementa il valore del semaforo di 1.

```
procedure wait (S)
begin
    if S > 0
        then S := S-1;
    else blocca il processo su S;
end;
```

```
procedure signal (S)
begin
    if qualche processo è bloccato su S
        then attiva un processo bloccato;
    else S := S+1;
end;
```

I semafori sono anche detti *semafori contatori* per le operazioni su S.

Notare che il semaforo può essere utilizzata sia per implementare la mutua esclusione che garantire l'accesso ad'una risorsa condivisa.

Ad esempio, supponiamo di avere n locazioni di memoria utilizzabili. Mi basta usare un semaforo associato a queste locazioni inizializzato ad n , ed esso terrà controllo del numero di locazioni libere in ogni momento.

Quando un processo vuole ottenere l'accesso a tale risorsa, richiama *wait* il quale verifica quante risorse sono rimaste libere (controllando il valore del semaforo).

Se ce ne sono di libere viene decrementato di 1 il numero di risorse.

Se invece non vi è alcuna risorsa libera il processo si blocca finché qualcun altro processo non rilascia una risorsa.

I processi rilasciano le risorse attraverso *signal*, il quale verifica se ci sono dei processi bloccati.

Se c'è qualche processo bloccato, risveglia uno di questi processi in modo che questo possa accedere alla risorsa.

Se non c'è alcun processo bloccato, allora segnala semplicemente che c'è un'ulteriore risorsa libera incrementando di 1 il numero di risorse.

Da un punto di vista **implementativo** consideriamo il semaforo come una **struttura** che contiene un *valore* ed una *lista* che ci consente di creare una coda dei processi che, nel momento in cui trovano il semaforo con valore = 0, si bloccano e attendono che tale valore diventi > 0.

La *wait* verifica se il valore del semaforo è > 0, in tal caso lo decrementa, altrimenti aggiunge il processo alla coda dei processi bloccati ed invoca una funzione per bloccarli.

La *signal* verifica se qualche processo del semaforo è bloccato, in tal caso toglie tale processo dalla coda dei processi bloccati e lo sveglia, altrimenti incrementa il valore del semaforo.

```
wait(semaphore *S) {
    if (S->value > 0)
        S->value--;
    else {
        aggiungi P a S->list;
        block();
    }
}
```

```
signal(semaphore *S) {
    if (qualche P bloccato su S){
        togli P da S->list;
        wakeup(P);
    }
    else
        S->value++;
}
```

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

nb:

Ci sono anche semafori che sono implementati CON valori negativi, i quali capiscono se un qualche processo è bloccato se nella *signal* il valore del semaforo è <= 0.

Le implementazioni sono equivalenti e un sistema potrebbe decidere di scegliere l'una o l'altra, anche se tutti i sistemi moderni utilizzano quelli a valori NON negativi.

Ricordiamo ulteriormente che tutte queste operazioni sono *atomiche*, altrimenti il loro discorso non avrebbe senso.

USO DEI SEMAFORI NEI SISTEMI CONCORRENTI

Uso	Descrizione
Mutua esclusione	La mutua esclusione può essere implementata usando un semaforo inizializzato a 1. Un processo effettua un'operazione <i>wait</i> sul semaforo prima di entrare in una CS e un'operazione <i>signal</i> al termine della CS. Uno speciale tipo di semaforo chiamato <i>semaforo binario</i> semplifica ulteriormente l'implementazione della CS.
Concorrenza limitata	La concorrenza limitata implica che una funzione può essere eseguita, o una risorsa può essere utilizzata, da n processi concorrentemente, $1 \leq n \leq c$, con c costante. Un semaforo inizializzato a c può essere utilizzato per implementare la concorrenza limitata.
Segnalazione	La segnalazione viene usata quando un processo P_i vuole effettuare un'operazione a_i solo dopo che il processo P_j ha effettuato una operazione a_j . Viene implementata utilizzando un semaforo inizializzato a 0. P_i effettua una <i>wait</i> sul semaforo prima di effettuare l'operazione a_i . P_j effettua una <i>signal</i> sul semaforo dopo aver effettuato l'operazione a_j .

USO: mutua esclusione

Supponiamo di voler implementare la mutua esclusione su due processi P_i e P_j . Usiamo un semaforo inizializzato ad 1. La prima cosa che deve fare il processo è invocare la *wait* sul semaforo ed entra in SC. Durante questo periodo di tempo l'altro processo sarà costretto ad aspettare (in quanto il valore del semaforo è 0). Uscito dalla SC il processo invoca la *signal* riportando il semaforo ad 1.

```

var sem_CS : semaforo := 1;
Parbegin
repeat
    wait (sem_CS);
    { Sezione critica }
    signal (sem_CS);
    { Parte restante del ciclo }
all'infinito;
Parend;
end.

Processo Pi                                Processo Pj

```

USO: concorrenza limitata

Supponiamo che possiamo avere fino a **c** processi che possono eseguire concorrentemente una certa operazione *op*.

Ogni processo che vuole eseguire tale operazione deve prima effettuare una *wait* sul semaforo, il quale sarà inizializzato al numero **c**, e poi deve effettuare una *signal* sullo stesso semaforo dopo averla eseguita.

USO: segnalazione tra processi

Supponiamo di avere un semaforo inizializzato a 0 e due processi P_i e P_j , dove P_i può eseguire la sua operazione a_i solo dopo che P_j ha eseguito la sua operazione a_j .

Per far ciò faremo innanzitutto invocare una *wait* da P_i , ma siccome il semaforo è 0 il processo verrà sicuramente bloccato. Quando verrà schedulato P_j , ed avrà effettuato la sua operazione a_j , invocherà la *signal* su quel semaforo e di conseguenza verrà risvegliato il processo P_i che adesso potrà effettuare la sua operazione a_i .

(Ovviamente se a_j viene eseguito prima di a_i il problema non si pone e P_i procede senza problemi).

Notare che non possono verificarsi *race condition* in quanto le operazioni *wait* e *signal* sono indivisibili; inoltre si tratta di un *semaforo binario* in quanto ha solo i valori 0 e 1.

```
var sync : semaforo := 0;
Parbegin
    ...
    wait (sync);
        { Esegui l'azione  $a_i$  }
    signal (sync);
    { Esegui l'azione  $a_j$  }
Parend;
end.
```

Processo P_i *Processo P_j*

NB: SEGNALI BINARI

I segnali binari sono una variante dei semafori il cui valore può assumere solo i valori 0 ed 1. La differenza è SOLO concettuale, infatti la semantica è la stessa (non sono un tipo diverso di semaforo).

Hanno lo scopo di semplificare la vita del programmatore, in quanto invece di dover ragionare con i contatori basterà usare gli 0 ed 1.

In particolar modo, se un processo chiama una *wait* e il valore è 1, il processo occupa la "risorsa" e il valore diventa 0, altrimenti si blocca.

Se un processo chiama una *signal* e la lista è vuota, pone il valore ad 1, altrimenti toglie il processo bloccato dalla lista e lo sveglia.

SOLUZIONI AI PROBLEMI DI SINCRONIZZAZIONE CON SEMAFORI

Produttori-Consumatori con buffer limitati (un solo buffer)

Per **semplicità** supponiamo che abbiamo un unico buffer. Dobbiamo fare in modo che il produttore non possa produrre finché il consumatore non ha consumato, e viceversa.

Utilizziamo **due semafori**:

- semaforo *full*: conta le locazioni piene;
- semaforo *empty*: conta le locazioni vuote;

Ovviamente all'inizio *full* sarà 0 ed *empty* sarà 1. Il buffer è rappresentato da un array di 1 elemento.

Il produttore, per produrre, deve assicurarsi che il buffer sia vuoto e quindi invocare la *wait* su *empty*. Se così fosse, va in SC e poi invoca la *signal* su *full* per indicare che ora il buffer è pieno.

Simmetricamente il consumatore, per consumare, deve assicurarsi che il buffer sia pieno e quindi invocare la *wait* su *full*. Se così fosse, va in SC e poi invoca la *signal* su *empty* per indicare che ora il buffer è vuoto.

```
type item = . . .;
var
    full : Semaphore := 0; { Initializations }
    empty : Semaphore := 1;
    buffer : array [0] of item;
begin
Parbegin
repeat
    wait (empty);
    buffer [0] := . . .;
    { i.e., produce }
    signal (full);
    { Remainder of the cycle }
forever;
Parend;
end.

Produttore
repeat
    wait (full);
    x := buffer [0];
    { i.e., consume }
    signal (empty);
    { Remainder of the cycle }
forever;
Consumatore
```

Tale soluzione evita l'attesa attiva poiché i semafori sono usati per controllare se i buffer sono pieni o vuoti.

La concorrenza totale nel sistema è 1, in quanto un solo processo per volta può accedere al buffer.

Produttori-Consumatori con buffer limitati (n buffer)

Si tratta di una **generalizzazione** della soluzione precedente. In questo caso abbiamo un array (circolare) di n buffer.

Anche in questo caso *full* sarà 0, tuttavia *empty* sarà n in quanto abbiamo n buffer.

Inoltre, visto che ho più buffer, avrò anche bisogno di due **indici**:

- *prod_ptr*: indice che usa il produttore per scorrere il buffer;
- *cons_ptr*: indice che usa il consumatore per scorrere il buffer;

Entrambi saranno, ovviamente, inizializzati a 0.

Nel momento in cui si va a produrre o consumare, deve anche essere aggiornato l'indice corrispondente così da indicare quale dovrà essere il nuovo buffer in cui si potrà produrre o consumare.

Per il resto, i semafori *empty* e *full* vengono utilizzati con la stessa logica di prima solo che ora dobbiamo tenere in considerazione più buffer.

```
const      n = . . .;
type       item = . . .;
var
            buffer : array [0..n - 1] of item;
            full : Semaphore := 0; { Initializations }
            empty : Semaphore := n;
            prod_ptr, cons_ptr : integer;
begin
            prod_ptr := 0;
            cons_ptr := 0;
Parbegin
repeat
            wait (empty);
            buffer [prod_ptr] := . . .;
            { i.e. produce }
            prod_ptr := prod_ptr + 1 mod n;
            signal (full);
            { Remainder of the cycle }
        forever;
Parend;
end.
```

Produttore Consumatore

OSS1: non è importante l'ordine di schedulazione. Un produttore può produrre finché ci sono buffer vuoti ed un consumatore può consumare finché ci sono buffer pieni.

OSS2: è anche possibile avere più produttori e consumatori, ma in tal caso l'aggiornamento degli indici corrispondenti deve avvenire in *mutua esclusione* altrimenti si ha *race condition*.

Lettori e Scrittori

Ricordiamo che i lettori possono leggere concorrentemente una certa risorsa, ma non possono leggere se qualcuno sta scrivendo sulla risorsa. Gli scrittori possono scrivere su una risorsa solo se non sta scrivendo già qualcun altro.

Per gestire questa situazione usiamo una serie di **contatori**:

- *runread*: numero di lettori in lettura;
- *runwrite*: numero di scrittori in scrittura;
- *totread*: numero di lettori che intendono leggere o in lettura;
- *totwrite*: numero di scrittori che intendono scrivere o in scrittura;

I contatori “tot” servono per implementare la soluzione con preferenza ai lettori/scrittori. Nel caso della preferenza ai lettori, usiamo *totread* per svegliare quei lettori che vogliono leggere ma non stanno leggendo.

Su queste variabili c’è *race condition*, quindi devono essere accedute in mutua esclusione.

Un lettore può leggere se non ci sono scrittori attivi, quindi controlla che *runwrite* ≠ 0. In tal caso legge, e dopo aver letto verifica se è stato l’ultimo che ha letto (*runread* = 0) e se ci sono scrittori che vogliono scrivere (*totwrite* ≠ 0); in tal caso attiva uno di questi scrittori.

Uno scrittore può scrivere se non ci sono lettori attivi e ANCHE se nessun altro scrittore sta scrivendo (quindi deve aspettare se *runread* ≠ 0 o *runwrite* ≠ 0).

In tal caso scrive, e dopo aver scritto verifica se ci sono lettori che vogliono leggere (*totread* ≠ 0) o scrittori che vogliono scrivere (*totwrite* ≠ 0); in tal caso attiva uno di questi scrittori o tutti i lettori (**nb**: la scelta dipende dal tipo di schema che vogliamo adottare).

Parbegin

repeat

if *runwrite* ≠ 0
then
 { attendi };
 { scrivi }
 if *runread* = 0 and
 totwrite ≠ 0
 then
 attiva uno scrittore in attesa

forever;
Parend;

Lettore/i

repeat

if *runread* ≠ 0 or
 runwrite ≠ 0
then { attendi };
 { scrivi }
 if *totread* ≠ 0 or *totwrite* ≠ 0
 then
 attiva uno scrittore in attesa o
 un lettore in attesa

forever;

Scrittore/i

Lettori e Scrittori (con preferenza ai lettori)

Utilizziamo **3 semafori**, 2 per la segnalazione, inizializzati a 0 :

- *reading*: per stabilire quando un processo può leggere;
- *writing*: per stabilire quando un processo può scrivere (o leggere).

ed 1 per la mutua esclusione, inizializzato ad 1:

- sem_CS;

Tutti i contatori sono ovviamente inizializzati a 0.

Quando un lettore vuole leggere deve innanzitutto entrare in mutua esclusione invocando una *wait* su sem_CS. Quando arriva il suo turno deve incrementare *totread*, e se nessuno sta scrivendo allora può leggere ed incrementa *runread* e invoca il *signal* su *reading*. Infine invoca una *signal* su sem_CS per indicare che esce dalla SC.

Poi invoca una *wait* su *reading* e va a leggere. Finito di leggere deve rientrare in SC con una *wait* su sem_CS e decrementare sia *runread* che *totread*.

Dopodichè controlla se è stato l'ultimo lettore e se ci sono scrittori in attesa, e in tal caso da la possibilità ad altri scrittori di andare a scrivere (*ruwrite* = 1 perché può scrivere un solo scrittore per volta).

Infine invoca una *signal* su sem_CS per indicare che esce dalla SC.

Una situazione più o meno simile avviene anche per lo scrittore, dove ovviamente deve prendere le sue giuste accortenze.

```

var
    totread, runread, totwrite, runwrite : integer;
        reading, writing : semaforo := 0;
        sem_CS : semaforo := 1;
begin
    totread := 0;
    runread := 0;
    totwrite := 0;
    runwrite := 0;
Parbegin
    repeat                                repeat
        wait (sem_CS);
        totread := totread + 1;
        if runwrite = 0 then
            runread := runread + 1;
            signal (reading);
            signal (sem_CS);
            wait (reading);
            { Leggi }
            wait (sem_CS);
            runread := runread - 1;
            totread := totread - 1;
            if runread = 0 and
                totwrite > runwrite
            then
                runwrite := 1;
                signal (writing);
                signal (sem_CS);
forever;                                forever;
Parend;
end.

```

Lettore/i

Scrittore/i

IMPLEMENTAZIONE DEI SEMAFORI

I semafori sono considerati come **strutture**, dove abbiamo: un *contatore*, una *lista* (dei processi che si bloccano), un variabile booleana *lock* (per realizzare l'atomicità dei *wait* e *signal*).

Dichiarazione del tipo semaforo

type

```
semaforo = record
    value : integer;      { valore del semaforo }
    list : ...            { lista dei processi bloccati }
    lock : boolean;       { variabile di lock per le operazioni su questo semaforo }
end;
```

Implementazione WAIT e SIGNAL

Devo garantire l'atomicità e il non verificarsi di race condition. All'interno del *wait* necessito di una funzione che mi indichi che è già in atto un'operazione sul semaforo, ovvero **chiudo** (acquisisco) **il lock**. Tale operazione è implementata a sua volta con *operazioni indivisibili* (che richiedono del busy wait, ma i tempi di attesa sono molto brevi).

Controllo se il valore del semaforo è >0, in tal caso ne decremento il valore e **apro** (rilascio) **il lock**, altrimenti aggiungo il processo alla **lista** dei processi bloccati e rilascio il lock (*block_me* blocca il P e rilascia il lock contemporaneamente).

La *signal* si comporta in modo simile: innanzitutto ascuisce il lock, poi controlla se c'è qualche processo bloccato e in caso lo attiva, altrimenti incrementa il contatore del semaforo. Solo al termine dei controlli, rilascio il lock.

Procedures for implementing wait and signal operations

```
procedure wait (sem)
begin
    Close_lock (sem.lock);
    if sem.value > 0
        then
            sem.value := sem.value-1;
            Open_lock (sem.lock);
        else
            Add id of the process to list of processes blocked on sem;
            block_me (sem.lock);
    end;
procedure signal (sem)
begin
    Close_lock (sem.lock);
    if some processes are blocked on sem
        then
            proc_id := id of a process blocked on sem;
            activate (proc_id);
        else
            sem.value := sem.value + 1;
            Open_lock (sem.lock);
    end;
```

Implementazioni di *wait* e *signal*:

- **Kernel-Level:** mediante system call
- **User-Level:** System calls solo per bloccare/attivare
- **Irido:** combinazione dei due

le implementazioni *wait* e *signal* sono ibride nei moderni SO, cioè sono chiamate utente che invocano a loro volta delle System Call

LEZIONE 9 – SINCRONIZZAZIONE DEI PROCESSI (4)

I MONITOR

I monitor sono un costrutto di sincronizzazione alternativo ai semafori, di *alto livello*, nati per semplificare l'implementazione di programmi concorrenti in quanto alleviano il carico di lavoro e responsabilità nell'implementare correttamente la mutua esclusione (se ne occupa il Monitor stesso).

Un **Monitor** è una raccolta di dati, strutture dati e procedure raggruppate in un modulo o pacchetto. Esso definisce un *lock* (implicito) e zero o più variabili di condizione per gestire l'accesso concorrente ai dati.

Per quanto riguarda la loro interazione con i processi:

- I processi possono invocare le procedure di un Monitor in ogni momento;
- I processi non possono accedere ai dati di un Monitor se non attraverso le procedure del Monitor;
- In un Monitor può essere attivo un solo processo alla volta (**mutua esclusione**).

Un monitor incapsula i dati da proteggere e acquisce il lock; opera su tali dati e rilascia il lock alla fine. Se non può completare rilascia temporaneamente il lock e lo riacquisisce appena può continuare.

Per l'elaborazione concorrente, un Monitor deve fornire *strumenti di sincronizzazione*, supportati dalle **variabili di condizione** le quali sono contenute ed accessibili solo dal Monitor stesso.

Esse sono un tipo speciale di dato sul quale sono possibili *due* operazioni:

- `cond_wait(c)`: sospende, sulla variabile di condizione *c*, l'esecuzione del processo che la invoca.
- `cond_signal(c)`: riprende l'esecuzione di un processo bloccato da una `cond_wait` sulla stessa variabile di condizione.

Dichiareremo le variabili condizioni con `condition x`;

OSSERVAZIONE:

Le operazioni `cond_wait()` e `cond_signal` sono diverse rispetto alle corrispondenti nei semafori, in quanto possono essere viste come “operazioni di segnalazioni ma senza memoria”.

Infatti con i semafori, il *wait* e il *signal* decrementano e incrementano un contatore mentre quelle dei monitor si limitano a bloccare e sbloccare i processi.

Quindi se un processo invoca una *signal* e non c'è nessun processo in attesa sulla variabile di condizione, il segnale è perso.

REGOLE PER LE VARIABILI DI CONDIZIONE

Dopo una cond_signal è necessario avere solo un processo attivo nel monitor, e quale processo viene risvegliato dipende dalla regola utilizzata:

- **Hoare**: viene eseguito il processo appena risvegliato, sospendendo quello che ha invocato la cond_signal;
- **Brich Hansen**: il processo che invoca cond_signal deve uscire immediatamente, ovvero cond_signal può apparire solo come istruzione finale di una procedura del monitor;
- **Mesa**: il processo che invoca cond_signal continua, e l'altro processo viene risvegliato solo dopo che quello che ha invoca la cond_signal è uscito dal monitor.

SEMAFORO BINARIO implementato con MONITOR

Creiamo un costrutto di tipo monitor *Sem_mon_type*.

Abbiamo bisogno di una variabile booleana *busy* che rappresenta il valore {0,1} del semaforo binario, ed una variabile di condizione *non_busy*.

In un *wait*, se *busy* è vera allora c'è già un altro processo in SC e quindi aspetta; altrimenti pone *busy = true* ed entra in SC.

In una *signal*, pone *busy = false* perché significa che il processo sta uscendo dalla SC, ed inoltre richiama la cond_signal della variabile di condizione *non_busy*.

La stesura del monito si conclude con il codice di inizializzazione, che in tal caso corrisponde semplicemente a porre *busy = false* in quanto inizialmente non c'è alcun processo in SC.

```
type Sem_Mon_type = monitor
var
    busy:boolean;
    non_busy: condition;
procedure sem_wait;
begin
    if busy = true then non_busy.cond_wait;
    busy := true;
end
procedure sem_signal;
begin
    busy := false;
    non_busy.cond_signal;
end
begin { initialization }
    busy := false;
end
```

var binary_sem : Sem_Mon_type; begin		
Parbegin		
repeat	repeat	repeat
binary_sem.sem_wait; binary_sem.sem_wait; binary_sem.sem_wait;		
{ Critical Section } { Critical Section } { Critical Section }		
binary_sem.sem_signal; binary_sem.sem_signal; binary_sem.sem_signal;		
{ Remainder of { Remainder of { Remainder of		
the cycle } the cycle } the cycle }		
forever;	forever;	forever;
Parend;		
end.		
	Process P ₁	Process P ₂
		Process P ₃

PRODUTTORI-CONSUMATORI CON MONITOR

Ci definiamo un Monitor che rappresenta il nostro buffer, il quale conterrà una costante n che indica la dimensione del buffer, una type *item* che indica gli elementi che è possibile produrre in tale buffer, ed altre variabili: l'array che contiene gli n elementi del buffer e gli indici per poter scorrere il buffer da parte dei produttori e dei consumatori.

Inoltre dichiariamo due variabili di *condizione*: *buff_full* e *buff_empty*.

Nell'inizializzazione, dichiareremo una variabile *full* = 0 con cui terremo conto dei buffer occupati, e inizializzeremo i due indici sempre a 0.

La procedura *produce* controlla innanzitutto se il buffer è pieno; se così fosse aspetta la *condizione* per cui almeno un buffer è vuoto, altrimenti produce ed incherà con una *signal* che è disponibile un buffer pieno.

La procedura *consume* controlla innanzitutto se il buffer è vuoto; se così fosse aspetta la *condizione* per cui almeno un buffer è pieno, altrimenti consuma ed indicherà con una *signal* che è disponibile un buffer vuoto (è simmetrica a *produce*).

```
type Bounded_buffer_type = monitor
  const
    n = . . .;                                { Numeri di buffer }
  type
    item = . . .;
  var
    buffer : array [0..n-1] of item;
    full, prod_ptr, cons_ptr : integer;
    buff_full : condition;
    buff_empty : condition;
  procedura produci (produced_info : item);
  begin
    if full = n then buff_empty.wait;
    buffer [prod_ptr] := produced_info;          { i.e. produci }
    prod_ptr := prod_ptr + 1 mod n;
    full := full + 1;
    buff_full.signal;
  end;
  procedura consuma (for_consumption : item);
  begin
    if full = 0 then buff_full.wait;
    for_consumption := buffer[cons_ptr];          { i.e. consuma }
    cons_ptr := cons_ptr + 1 mod n;
    full := full - 1;
    buff_empty.signal;
  end;
  begin { inizializzazione }
    full := 0;
    prod_ptr := 0;
    cons_ptr := 0;
  end;
```

PROBLEMA DEL BARBIERE ADDORMENTATO (classico prob. di sinc)

Tale problema riguarda un negozio dove lavora UN barbiere, ci sono N sedie per i clienti in attesa ed UNA poltrona da barbiere.

Specifiche:

- se non ci sono clienti, il barbiere si addormenta sulla poltrona;
- quando arriva, un cliente sveglia il barbiere e si fa tagliare i capelli sulla poltrona;
- se arriva un cliente mentre il barbiere sta tagliando i capelli ad un altro cliente, il cliente attende su una delle sedie libere;
- se tutte le sedie sono occupate, il cliente se ne va.

Avrò bisogno di un semaforo `sem_CS` inizializzato ad 1 (perché la prima volta lo voglio usare) per la mutua esclusione. Dichiarerò inoltre altri due semafori: `ClienteDisponibile` e `Poltrona` per indicare rispettivamente la disponibilità del cliente e della poltrona.

`inAttesa` indica il numero di clienti in attesa.

`Sedie` indica il numero di sedie totali.

Il **barbiere** controlla se ci sono clienti disponibili, quindi fa una `wait` su `ClienteDisponibile`. Se ce ne sono, chiama un cliente a tagliare i capelli, quindi va in mutua esclusione facendo una `wait` su `sem_CS` e decrementa `inAttesa`.

Faccio poi una `signal` su `Poltrona` così che un cliente che stava aspettando di sedersi sulla poltrona, possa sedersi. Poi rilascio la mutua esclusione.

Infine il barbiere si mette a tagliare i capelli.

Un **cliente** che arriva in negozio controlla innanzitutto che ci siano sedie libere (ovvero, controlla che il numero di clienti in attesa sia minore delle sedie disponibili).

In tal caso, aumenta `inAttesa`.

Faccio poi una `signal` su `ClienteDisponibile` per indicare al barbiere che c'è un nuovo cliente, e quest'ultimo fa una `wait` su `Poltrona` per indicare che è occupata. Poi rilascio la mutua esclusione.

Infine il cliente si taglia i capelli.

Se non ci fossero posti disponibili, il cliente rilascia la mutua esclusione e se ne va.

```

sem_CS: semaphore := 1;
clienteDisponibile, poltrona : semaphore := 0;
inAttesa : integer;
SEDIE : const;

begin
  SEDIE := N;
  inAttesa := 0;

Parbegin
  repeat

    wait(ClienteDisponibile);
    wait(sem_CS);
    inAttesa := inAttesa - 1;
    signal(poltrona);
    signal(sem_CS);
    {taglia capelli};

    wait(sem_CS);
    if inAttesa < SEDIE then
      inAttesa := inAttesa + 1;
      signal(ClienteDisponibile);
      signal(sem_CS);
      wait(poltrona);
      {taglio dei capelli}

  else signal(sem_CS);
  {lascia il negozio};

Parend
Barbiere Clienti

```

nb: I Monitor sebbene siano più comodi, sono disponibili solo per un numero ristretto di linguaggi (es. Java, non disponibili in C o C++). Negli altri casi dovremmo costruirli da 0, creando ad esempio una classe (oppure usiamo i semafori).

CASI DI STUDIO DI SINCRONIZZAZIONE DI PROCESSI

Sincronizzazione dei Thread POSIX

I thread POSIX forniscono:

- Mutex (sono come semafori binari);
- Semafori;
- Varabili di condizione.

Un SO può implementare i thread POSIX come thread di livello kernel o thread di livello utente.

Sincronizzazione di Processi in UNIX

UNIX mette a disposizione due tipi di semafori:

- semafori POSIX, dalla libreria pthread POSIX (mantenuti nel File System);
- semafori System V (mantenuti nel kernel)..

Sincronizzazione di Processi in Linux

Linux ha semafori Unix-like per i processi utente, ma fornisce anche semafori usati dal kernel.

LEZIONE 11 – MESSAGE PASSING

Lo **scambio dei messaggi**, o message passing, è una strategia di comunicazione e sincronizzazione che può essere adottata per quelle applicazioni distribuite, ovvero per quei processi che giacciono su host diversi.

Tale funzionalità è fornita mediante una coppia di primitive:

- send (destination,message): un processo invia delle informazioni ad un altro processo designato come destinazione;
- receive(destination,message): un processo riceve delle informazioni da un altro processo designato come sorgente.

Per poter comunicare, è di fondamentale importanza la sincronizzazione fra questi processi.

Quando è eseguita una **send**:

- il processo mittente viene bloccato fintanto che il messaggio inviato non è stato ricevuto;
- il processo mittente continua anche se il messaggio inviato non è stato ricevuto.

Quando è eseguita una **receive** possono accadere due cose:

- se il messaggio da ricevere è stato già inviato, allora tale messaggio viene ricevuto dal processo che ha invocato la receive e continua;
- se il messaggio da ricevere non è stato ancora inviato, allora il processo che ha invocato la receive viene bloccato finché tale messaggio non viene inviato; oppure il processo continua l'esecuzione però rinuncia alla volontà di ricerverlo.

COMBINAZIONI DI SEND e RECEIVE

Send e Receive bloccanti: sono primitive dove sia il mittente che il destinatario sono bloccati fintantoché il messaggio non è consegnato.

Questo tipo di comunicazione prende il nome di *rendezvous*, e permette una sincronizzazione stretta tra processi.

Send non bloccante, Receive bloccante: sono primitive dove il mittente è bloccato fino a che il messaggio richiesto arriva.

Si tratta della combinazione più utile, ed invia uno o più messaggi a più destinazioni il più velocemente possibile.

Tuttavia, in caso di errori possono presentarsi situazioni in cui il processo invia continuamente messaggi (consumando risorse). Inoltre è il programmatore che deve assicurarsi che il messaggio sia ricevuto.

Send non bloccante, Receive non bloccante: a nessuna delle controparti è richiesto di attendere.

INDIRIZZAMENTO

Gli schemi per specificare i processi nelle primitive *send* e *receive* ricadono in due categorie: *indirizzamento diretto* e *indirizzamento indiretto*.

Nella tecnica di denominazione diretta, i processi mittente e destinatario dichiarano il proprio nome, mentre nella tecnica di denominazione indiretta questo non avviene.

L'indirizzamento indiretto è molto più flessibile in quanto viene disaccoppiato il processo mittente da quello destinatario, e sfrutta delle *mailbox*.

MAILBOX

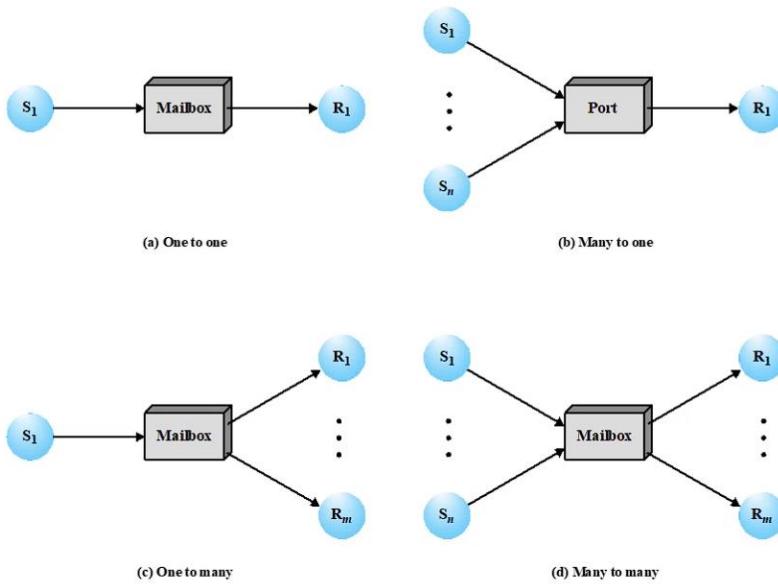
Le Mailbox sono delle sorta di deposito per messaggi interprocesso. Una Mailbox ha un nome unico e il proprietario è di solito il processo che l'ha creata.

Si tratta di un indirizzamento **indiretto** in quanto solo il proprietario può ricevere messaggi, e se dei processi devono inviare un messaggio a tale processo, lo inviano alla mailbox (ovviamente, ne devono conoscere il nome)

Il kernel può supportare le mailbox in diversi modi: può ad esempio fornir loro un insieme fissato di nomi, o può permettere di far ciò ai processi utente.

Esistono 4 tipi di **relazione** fra i processi mittenti e quelli destinatari che comunicano attraverso Mailbox:

- **Uno a uno**: sono le tipiche comunicazioni *private* fra due processi;
- **Molti ad uno**: sono le tipiche comunicazioni *client/server*, dove abbiamo n client ed 1 server, e dove la mailbox prende il nome di *porta*.
- **Uno a Molti**: sono le tipiche comunicazioni in *broadcast*.
- **Molti a Molti**: sono le tipiche comunicazioni fra *client/server*, dove abbiamo n client ed n server.



ASSOCIAZIONE E PROPRIETÀ DELLE MAILBOX

Il tipo di associazione assegnato ad una mailbox può essere:

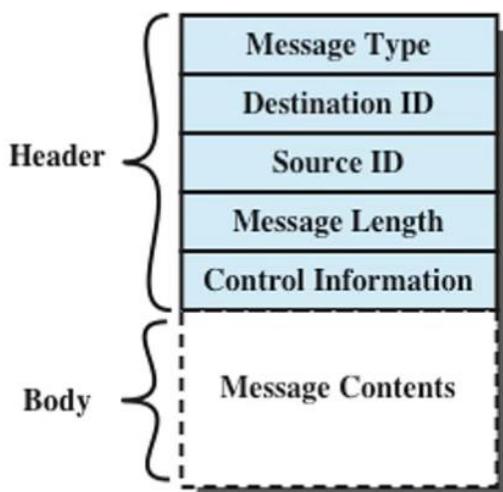
- **statica**: l'associazione è creata in modo permanente al processo che crea l'associazione. È il caso delle associazioni N:1 e 1:1.
- **dinamica**: prima di creare l'associazione si deve specificare a quale mailbox ci si deve connettere; implica l'uso di primitive di tipo *connect* e *disconnect*. È il caso delle associazioni 1:N e M:N.

Quando il processo che crea la mailbox **termina**, nei processi N:1 la porta (cioè la mailbox) è eliminata. Negli altri casi il SO può permettere sia che la mailbox siano eliminate quando termina il processo, oppure queste devono essere eliminate tramite un comando speciale.

FORMATO DEI MESSAGGI

Può essere di vari tipi e dipende sia dal meccanismo di scambio dei messaggi, che dal fatto che sia eseguito in locale o su un sistema distribuito.

In sostanza, possiamo distinguere messaggi di *lunghezza fissa* e di *lunghezza variabile*.
esempio di mex. var:



MUTUA ESCLUSIONE con SCAMBIO DI MESSAGGI

Lo scambio dei messaggi può essere utilizzato per la mutua esclusione.

In questo tipo di soluzione si suppone di avere una combinazione *send non bloccante*, *receive bloccante*.

Si crea una mailbox condivisa fra tutti i processi che devono entrare in SC, la quale è inizializzata a *null*.

Un processo che vuole entrare in SC dovrà prima ricevere un messaggio. Avvenuto ciò, uscito dalla SC, prima di continuare la sua esecuzione, reinvia il messaggio nella Mailbox con una *send*.

Il messaggio sarà presente nella mailbox quando ci sarà un processo che avrà invocato una *send*. Se tale messaggio non c'è, il processo che voleva riceverlo verrà bloccato finché non sarà inserito un nuovo messaggio.

Proprio perché è possibile risvegliare un solo processo per volta, possiamo utilizzare i messaggi per implementare la mutua esclusione.

```
/* program mutual exclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

PRODUTTORE-CONSUMATORE con SCAMBIO MESSAGGI

Possiamo utilizzare la logica della *mutua esclusione con scambio di messaggi* per implementare una soluzione al problema del **Produttore-Consumatore**.

Con i semafori, se avevamo un solo produttore e consumatore, utilizzavamo un semaforo per la produzione ed uno per la consumazione.

In modo simile utilizzeremo una mailbox per la produzione ed una per la consumazione.

Ciò che produce il produttore è un messaggio inviato alla mailbox *mayconsume*, intendendo che il consumatore può consumare. Quando consumatore riceve il messaggio e lo consuma, invierà a sua volta un messaggio alla mailbox *mayproduce*, intendendo che il produttore può produrre.

I dati all'interno di questo buffer sono organizzati come una *coda* di messaggi, cioè il primo messaggio che viene inviato sarà anche il primo ad essere estratto ed elaborato.

Il buffer avrà una certa capacità *capacity*, e la mailbox *mayproduce* è riempita con un numero di messaggi *null* pari alla capacità del buffer. All'inizio il produttore potrà quindi sicuramente produrre.

nb: la receive è bloccante, altrimenti le cose non funzionerebbero.

```
const int
    capacity = /* buffering capacity */ ;
    null /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

STRATEGIE DI ACCODAMENTO

La strategia di accodamento più semplice è del tipo **FIFO**, ma è inappropriata se si vuole dare priorità all'importanza di un messaggio più che all'ordine.

Per tal motivo esistono delle **alternative**:

- Permettere di specificare una priorità per i messaggi sulla base del tipo di messaggio o come indicato dal mittente;
- Permettere al destinatario di ispezionare la coda dei messaggi e selezionare il messaggio successivo da ricevere.

SCAMBIO DEI MESSAGGI NEI VARI SO

Unix

Prevede tre supporti alla comunicazione interprocesso:

- **Pipe**: sono un meccanismo FIFO attraverso il quale i processi che giacciono nella stessa macchina possono comunicare tra loro, e ne esistono di due tipi: *con nome* e *anonime*.
- **Code di messaggi** (message queue);
- **socket**: sono un meccanismo utilizzate per le applicazioni di rete.

Windows

Prevede quattro supporti alla comunicazione interprocesso:

- **Pipe** con nome;
- **Local Procedure Call (LPC)**: scambio di messaggi tra processi nello stesso host.
- **Socket Windows (Winsock)**;
- **RPC**: sincrono e asincrono.

LEZIONE 12 – SCHEDULING (1)

Lo scheduling, in generale, è l'attività di selezione della successiva richiesta che deve essere elaborata da un server.

In un SO, una *richiesta* è l'esecuzione di un job o di un processo e il *server* è la CPU.

Possiamo distinguere diversi **tipi** di scheduling:

- **lungo termine**: decide quale processo deve essere ammesso all'insieme dei processi da eseguire;
- **medio termine**: decide quale processo deve essere presente totalmente o parzialmente in memoria (ovvero, quali processi porre in swap-in/swap-out);
- **breve termine**: decide quale processo è assegnato alla CPU;
- **I/O**: decide quale richiesta di I/O pendente da un processo deve essere gestita da un dispositivo di I/O disponibile.

Scheduling e Transizioni di stato

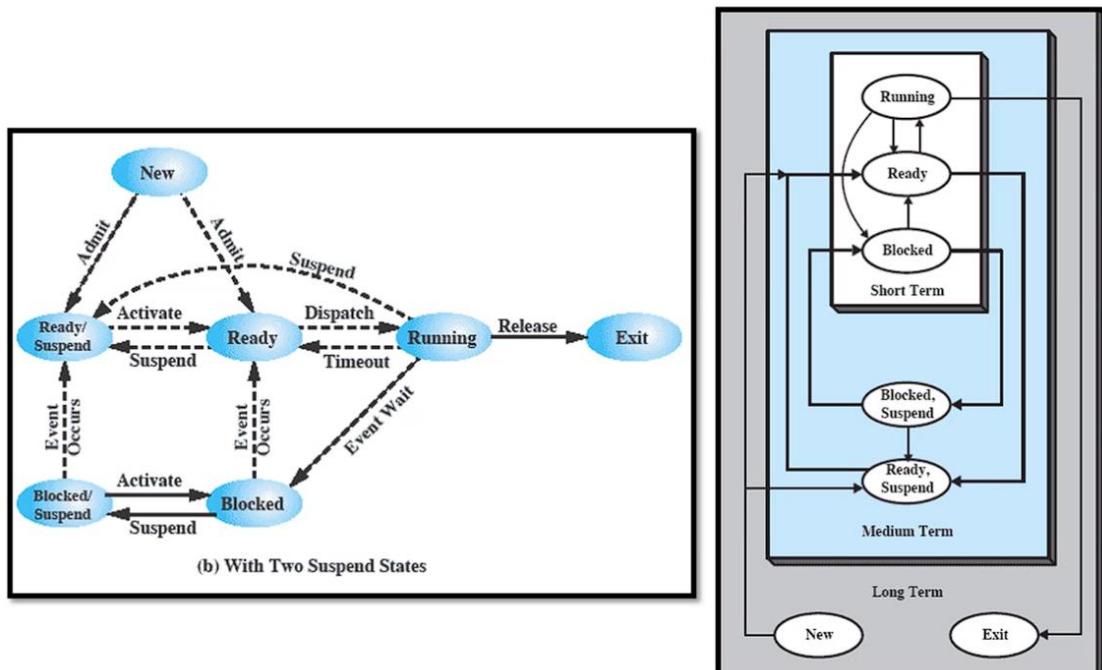
Lo scheduler a **lungo termine** decide quali processi devono essere portati in memoria, ed essi saranno poi schedulati dallo scheduler a *medio* o *breve termine*.

In base al sistema, un processo può partire dallo stato *ready* o *ready/suspend*.

Lo scheduler a **medio termine** si occupa di far passare un processo dallo stato *ready/suspend* a quello *ready*. Inoltre, si occuperà anche di portare i processi dallo stato *blocked/suspend* a quello *blocked* o a quello *ready/suspend*.

Lo scheduler a **breve termine** si occupa di far passare un processo dallo stato *ready* a quello *running* (e viceversa) o in quello *ready/suspend*. Inoltre, si occuperà anche di portare i processi dallo stato *blocked* a quello *blocked/suspended*.

Un processo viene caricato o tolto dalla memoria con le operazioni di swap-in e swap-out.

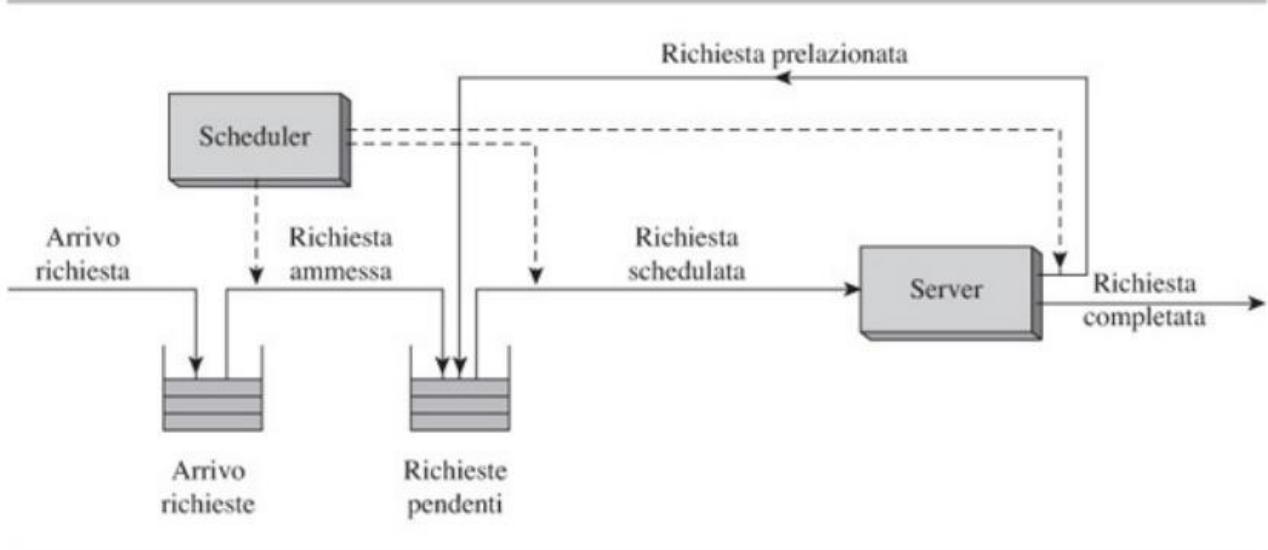


Schematizzazione dello scheduling

Le richieste in arrivo vengono inserite nell'insieme delle *richieste arrivate*, in attesa che vengano spostate dallo scheduler a lungo termine nell'insieme delle *richieste pendenti* (sono in memoria, sono quelle pronte per essere selezionate per l'esecuzione).

Lo scheduler a breve termine sposta le richieste pendenti alla CPU (server) in modo che possano essere elaborate.

Un processo dalla CPU può però essere prelazionato.



TERMINOLOGIA E CONCETTI DI SCHEDULING

Termine o concetto	Definizione o descrizione
Relativamente alla richiesta	
Tempo di arrivo	Istante in cui un utente invia un job o un processo.
Tempo di ammissione	Istante in cui il sistema comincia a considerare un job o un processo per lo scheduling.
Tempo di completamento	Istante in cui un job o un processo è terminato.
Deadline	Istante entro il quale un job o un processo deve essere terminato per rispettare il requisito di risposta di un'applicazione real-time.
Tempo di servizio	Il totale del tempo di CPU e quello di I/O richiesto da un job, processo o sottorichiesta per completare la sua operazione.
Preelezione	Deallocation forzata della CPU da un job o da un processo.
Priorità	Una regola discriminante usata per selezionare un job o un processo quando molti job o processi attendono l'elaborazione.
Relativamente al servizio per l'utente: richieste individuali	
Superamento della deadline (deadline overrun)	La quantità di tempo per la quale il tempo di completamento di un job o un processo supera la sua deadline. Il deadline overrun può essere sia positivo sia negativo.
Condivisione equa	Una condivisione specifica del tempo di CPU che dovrebbe essere dedicato all'esecuzione di un processo o di un gruppo di processi.
Rapporto di risposta	Il rapporto $\frac{\text{tempo di attesa} + \text{tempo di servizio di un job o processo}}{\text{tempo di servizio del job o processo}}$
Tempo di risposta (rt)	Il tempo tra la sottomissione di una sottorichiesta per l'elaborazione e il tempo in cui il suo risultato diventa disponibile. Questo concetto è applicabile ai processi interattivi.
Tempo impiegato per il completamento o tempo di turnaround (ta)	Il tempo che intercorre tra la sottomissione di un job o processo e il suo completamento da parte del sistema. Questo concetto è significativo solo per job non interattivi o processi.
Turnaround pesato (w)	Rapporto del tempo di turnaround di un job o processo e il suo tempo di servizio.
Relativamente al servizio per l'utente: servizio medio	
Tempo di risposta medio (\bar{rt})	La media dei tempi di risposta di tutte le sottorichieste elaborate dal sistema.
Tempo medio di turnaround (\bar{ta})	La media dei tempi di turnaround di tutti i job o processi elaborati dal sistema.
Relativamente alle prestazioni	
Durata della schedulazione	Il tempo necessario per completare un insieme specifico di job o processi.
Throughput	Il numero medio di job, processi o sottorichieste completate da un sistema nell'unità di tempo.

TECNICHE DI SCHEDULING

Gli scheduler usano **tre tecniche** fondamentali per ottenere un buon servizio utente ed elevate prestazioni di sistema:

- **Scheduling basato su priorità:** fornisce un elevato throughput del sistema. In particolar modo, si preferisce dare una maggior priorità ai sistemi I/O bound in quanto utilizzano poca CPU (e possono essere completati prima);
- **Riordino delle richieste:** attraverso la prelazione si stabiliscono dei criteri che consentono di avere un miglior riordino delle richieste. Infatti, senza prelazione, le richieste verrebbero eseguite nell'ordine in cui arrivano. In particolar modo, si preferisce eseguire prima i processi che richiedono un minor tempo di servizio così da migliorare il throughput.
- **Variazione dello slot temporale:** invece di dividere equamente gli slot temporali, questi variano in base alle caratteristiche dei processi. Notare che valori più piccoli degli slot temporali forniscono migliori tempi di risposta, ma anche una minore efficienza della CPU. Per bilanciare i *tempi di risposta* con le *prestazioni del sistema* si preferisce dunque assegnare slot più piccoli ai processi I/O bound e slot più grandi a quelli CPU bound.

ricordo:

$$rt = n \times (\delta + \sigma)$$

$$\eta = \delta / (\delta + \sigma)$$

Dove

n : numero utenti che usano il sistema,

δ : tempo richiesto per completare una sotto-richiesta,

σ : overhead dello scheduling,

η : efficienza di CPU

IL RUOLO DELLE PRIORITÀ

La priorità può essere:

- *statica*: dipende dalla tipologia del processo o dal carico del sistema;
- *dinamica*: dipende da una serie di criteri che devono essere impostati.

Sulla base delle priorità si può eseguire un riordino in base alle strategie adottate: ad esempio, eseguire i processi più brevi e poi quelli più lunghi.

Alcuni riordini invece necessitano di funzioni di priorità più complesse: in caso dei processi avessero la stessa priorità si può usare uno scheduling *round-robin*.

Lo scheduling che fa uso di *priorità* può portare a *starvation*, cioè c'è il pericolo che le richieste a più bassa priorità non vengano mai eseguita.

In tal caso si può usare l'*aging*, il quale aumenta la priorità di un processo all'aumentare del tempo trascorso.

SCHEDULING SENZA PRELAZIONE

In uno scheduling senza prelazione, la CPU serve sempre una richiesta schedulata fino al completamento.

Ha il vantaggio di essere estremamente semplice da realizzare, e un'eventuale scelta del processo da eseguire può essere fatta non solo sull'ordine di arrivo ma anche su altri aspetti.

Infatti, possiamo distinguere diverse **politiche** di scheduling senza prelazione:

- Scheduling First-Come, First-Served (**FCFS**);
- Scheduling Shortest Job First (**SJF**);
- Scheduling High Response Ratio (**HRN**);

Scheduling FCFS

Le richieste sono schedulate nell'ordine in cui giungono al sistema. La lista delle richieste in attesa è organizzata come una coda e lo scheduler seleziona sempre la prima richiesta nella lista.

Scheduling SJF

Le richieste sono schedulate dando priorità a quelle con il minimo tempo di servizio. Così, una richiesta rimane in attesa finché non siano state elaborate tutte le richieste più brevi.

Può causare *starvation* dei processi più lunghi. (MEGLIO PRIMA)

Scheduling HRN

Le richieste sono schedulate dando priorità a quelle con il più elevato *rapporto di risposta*, il quale è calcolato come:

$$\text{Rapporto di risposta} = \frac{\text{tempo di attesa} + \text{tempo di servizio del processo}}{\text{tempo di servizio del processo}}$$

L'uso di tale rapporto contrasta la *starvation*.

CONFRONTO FRA GLI ALGORITMI DI SCHEDULING SENZA PRELAZIONE

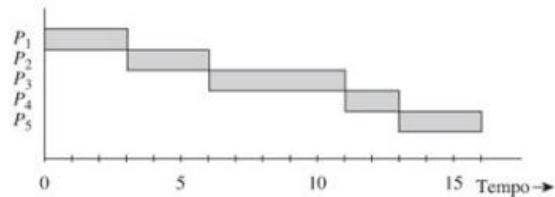
Tenendo in considerazione il seguente caso:

Processo	P_1	P_2	P_3	P_4	P_5
Tempo di ammissione	0	2	3	4	8
Tempo di servizio	3	3	5	2	3

Gli **strumenti** che utilizziamo per confrontare tali algoritmi sono il tempo di *turnaround* **ta** e il tempo di *turnaround* pesato **w**.

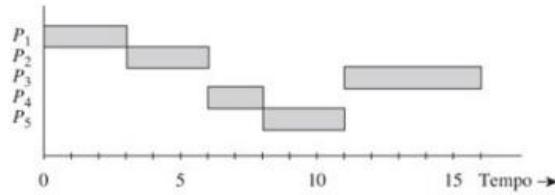
FCFS: si verifica una variazione considerevole nei completamenti pesati forniti. Questa variazione è maggiore nel caso in cui i processi soggetti a grandi di completamento sono brevi.

Tempo	Processo completato			Processi nel sistema (in ordine FCFS)	Processo schedulato
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	—	—	—	$\{P_1\}$	P_1
3	P_1	3	1.00	$\{P_2, P_3\}$	P_2
6	P_2	4	1.33	$\{P_3, P_4\}$	P_3
11	P_3	8	1.60	$\{P_4, P_5\}$	P_4
13	P_4	9	4.50	$\{P_5\}$	P_5
16	P_5	8	2.67	—	—



SJF: il tempo di completamento medio e il completamento pesato medio sono migliori rispetto allo scheduling FCFS poiché le richieste brevi vengono completate prima. Tuttavia le richieste più lunghe rischiano lo starvation.

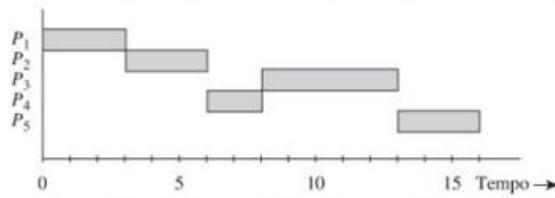
Tempo	Processo completato			Processi nel sistema (in ordine FCFS)	Processo schedulato
	<i>id</i>	<i>ta</i>	<i>w</i>		
0	—	—	—	$\{P_1\}$	P_1
3	P_1	3	1.00	$\{P_2, P_3\}$	P_2
6	P_2	4	1.33	$\{P_3, P_4\}$	P_4
8	P_4	4	2.00	$\{P_3, P_5\}$	P_5
11	P_5	3	1.00	$\{P_3\}$	P_3
16	P_3	13	2.60	{} —	—



HRN: Il rapporto di risposta del processo giunto per ultimo è 1. Esso si incrementa secondo il rapporto ($1/\text{tempo di servizio}$) mentre è in attesa di essere eseguito. Il rapporto di risposta di un processo breve si incrementa più rapidamente di quello di un processo lungo, quindi i processi più brevi sono privilegiati per quanto riguarda lo scheduling. Comunque, il rapporto di risposta di un processo lungo può eventualmente diventare

sufficientemente grande da consentire che il processo sia schedulato. Questa caratteristica provoca un effetto simile alla tecnica di aging prevenendo lo *starvation*.

Tempo	Processo completato			Response ratio dei processi					Processo schedulato
	<i>id</i>	<i>ta</i>	<i>w</i>	P_1	P_2	P_3	P_4	P_5	
0	—	—	—	1.00					P_1
3	P_1	3	1.00		1.33				P_2
6	P_2	4	1.33			1.00			P_4
8	P_4	4	2.00				2.00		P_5
13	P_3	10	2.00					1.00	P_3
16	P_5	8	2.67					2.67	—



POLITICHE DI SCHEDULING CON PRELAZIONE

Nello scheduling con prelazione, la CPU (server) può commutare alla prossima richiesta prima di completare quella corrente.

La richiesta prelazionata è messa in una lista di richieste pendenti, e il suo servizio è ripristinato quando è nuovamente schedulata.

Una richiesta tuttavia può anche essere schedulata molte volte prima che sia completata, e ciò comporta un maggior *overhead* rispetto allo scheduling senza prelazione.

È usato nei SO multi-programmati e time-sharing.

Scheduling Round-Robin (RR)

L'obiettivo del Round-Robin, più che essere posto sulle prestazioni del sistema (throughput), è quello di fornire **buoni tempi di risposta a tutte le richieste**.

Il **time slice δ** è la massima quantità di tempo di CPU che una richiesta schedulata può usare. Al termine di questo tempo, la richiesta viene sospesa (viene generato un'interrupt dal kernel), ovvero viene prelazionata IN coda della coda dei *processi pronti*.

Tale schema fornisce dei tempi di risposta comparabili a tutti i processi CPU-bound, ovvero se facciamo una valutazione noteremo che i tempi di turnaround pesati sono grosso modo uguali.

L'effettivo valore del turnaround pesato di un processo dipende dal numero di processi nel sistema.

I turnaround pesati di processi che fanno operazioni di I/O dipendono dalla durata di tali operazioni.

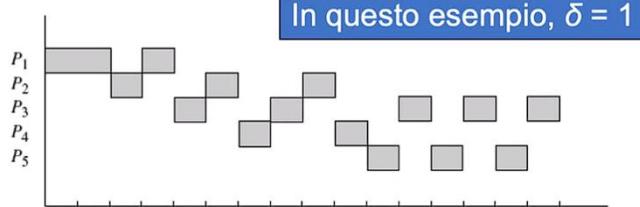
Lo scheduling RR non favorisce i processi brevi, e quindi non è una misura adeguata per le prestazioni del sistema.

Time of scheduling	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	c	t_a	w
Position of Processes in ready queue (1 implies head of queue)	P_1	1	1	2	1												4	4	1.33
P_2			1	3	2	1	3	2	1								9	7	2.33
P_3				2	1	3	2	1	4	3	2	1	2	1	2		16	13	2.60
P_4					3	2	1	3	2	1							10	6	3.00
P_5									3	2	1	2	1	2	1		15	7	2.33
Process scheduled	P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5				

$$\bar{t}_a = 7.4 \text{ seconds}, \bar{w} = 2.32$$

c: completion time of a process

In questo esempio, $\delta = 1$



Process	\varnothing	P_1	P_2	P_3	P_4	P_5	0	5	10	15	Time →
Admission time	0	2	3	4	8						
Service time	3	3	5	2	3						

Per semplicità abbiamo affermato che il tempo di risposta $rt = n \times (\delta + \sigma)$, dove n è il numero di processi nel sistema.

In realtà la relazione tra rt e δ è più **complessa**, infatti:

- l' rt in realtà è governato più dal numero di processo attivi che da n , infatti alcuni processi saranno bloccati per le operazioni di I/O o per l'attesa di azioni degli utenti;
- più saranno piccolo i valori di δ più saranno elevati i tempi di risposta, infatti se una richiesta ha bisogno di più δ secondi di tempo di CPU, sarà schedulata più di una volta prima di produrre una risposta.

Least Completed Next (LCN)

Schedula il processo che ha usato fino a quel momento la **minor quantità di tempo di CPU**, indipendentemente dalla natura del processo (CPU bound o I/O bound).

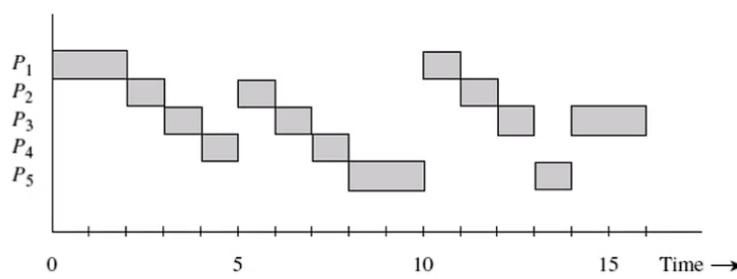
Tutti processi progrediscono in modo approssimativamente **uguale in termini di tempo** di CPU consumato, e quindi è garantito che i processi brevi finiscano prima dei processi lunghi (migliori prestazioni in quanto maggior throughput).

L'inconveniente è che i processi lunghi possono soffrire di *starvation*, ma anche i processi non molto lunghi o con elevati tempi di turnaround possono soffrire dello stesso problema in quanto LCN trascura i processi esistenti a vantaggio dei processi appena arrivati (in quanto questi ultimi saranno sicuramente quelli che hanno usato meno CPU).

Time of scheduling	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CPU time consumed by processes	P_1	0	1	2	2	2	2	2	2	2	2					
	P_2			0	1	1	1	2	2	2	2	2				
	P_3				0	1	1	1	2	2	2	2	2	3	4	5
	P_4					0	1	1	1							
	P_5								0	1	2	2	2	2		
Process scheduled	P_1	P_1	P_2	P_3	P_4	P_2	P_3	P_4	P_5	P_5	P_1	P_2	P_3	P_5	P_3	P_3

$$\bar{t}_a = 8.8 \text{ seconds}, \bar{w} = 2.72$$

c: completion time of a process



Process	P_1	P_2	P_3	P_4	P_5
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

I tempi rispetto al RR sono *peggiori* proprio perché si favoriscono sempre i processi appena arrivati mentre il RR cerca di trattare tutti allo stesso modo.

Shortest Time to Go (STG)

Schedula il processo che il minor requisito di tempo di CPU **restante**.

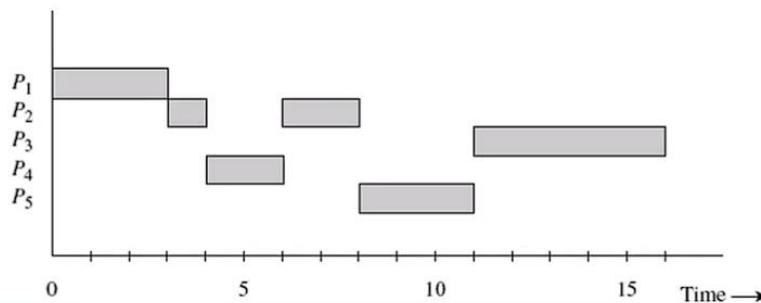
Favorisce i processi brevi e fornisce un buon throughput, e favorisce i processi prossimi ad essere completati (risolve il problema in cui i processi più vecchi sono trascurati a favore dei processi appena arrivati della LCS).

Il problema è che i processi lunghi possono soffrire di *starvation*.

Time of scheduling	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Remaining CPU time requirement of a process	P_1	3	2	1												
P_2				3	3	2	2	2	1							
P_3					5	5	5	5	5	5	5	5	4	3	2	1
P_4						2	1									
P_5									3	2	1					
Process scheduled	P_1	P_1	P_1	P_2	P_4	P_4	P_2	P_5	P_5	P_5	P_3	P_3	P_3	P_3		

$$\bar{t}_d = 5.4 \text{ seconds}, \bar{w} = 1.52$$

c: completion time of a process



Analogo alla politica SJF -> processi più lunghi possono andare in starvation

Process	P_1	P_2	P_3	P_4	P_5
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

LEZIONE 13 – SCHEDULING (2)

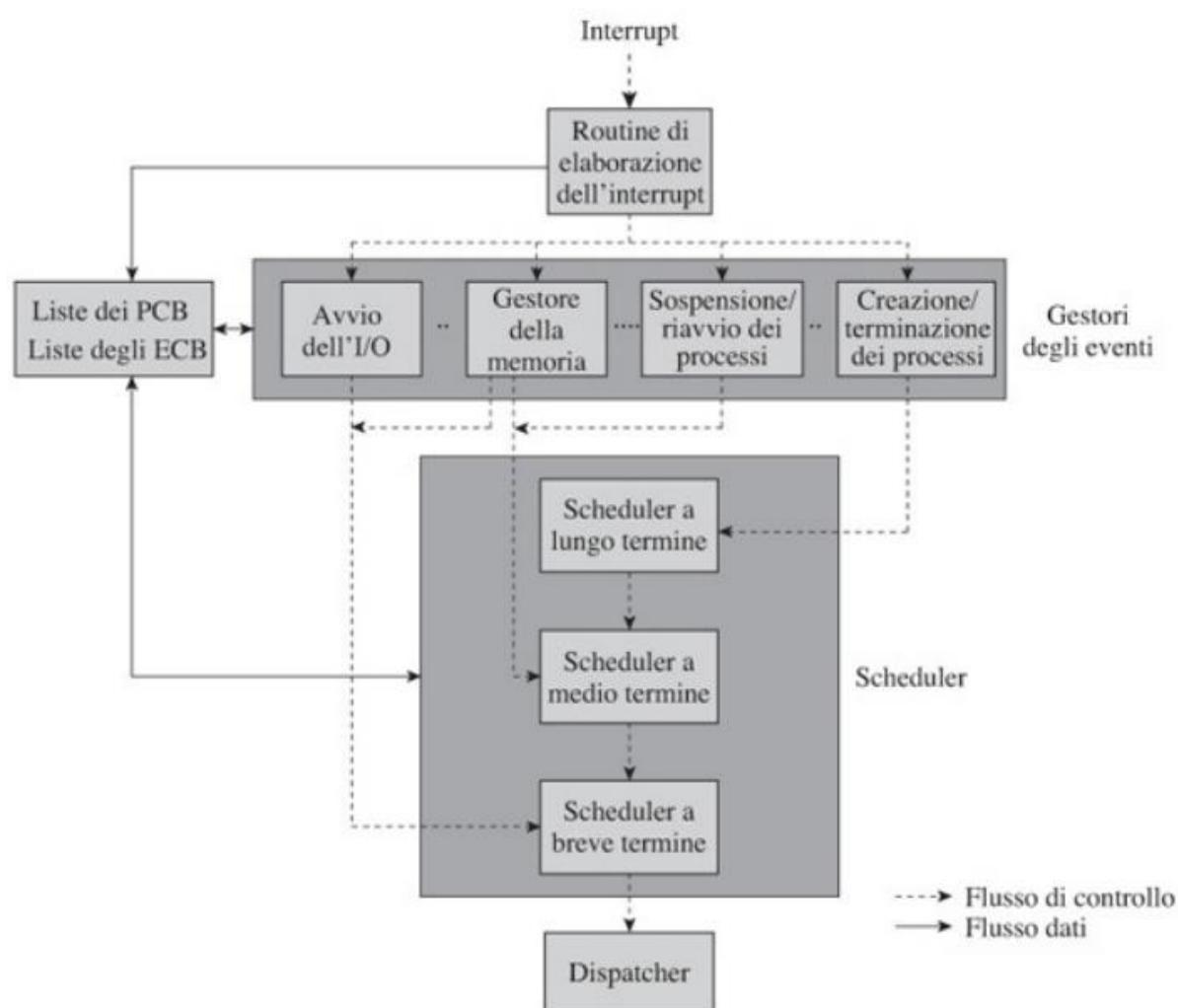
Per fornire una combinazione appropriata di prestazioni di sistema e servizio utente, il SO deve adattare il suo funzionamento alla natura e al numero delle richieste utente, e alla disponibilità delle risorse.

Un singolo scheduler che usa una strategia di scheduling classica non può affrontare in modo efficace tutti questi problemi, per tal motivo i SO moderni impiegano **più scheduler** (fino a tre).

Gli scheduler utilizzabili sono tre: a *lungo termine*, a *medio termine* e a *breve termine*.

Gestione degli eventi e Scheduling

Il funzionamento del kernel è interrupt-driven. Ogni evento che richiede l'attenzione del kernel causa un interrupt. La routine di elaborazione dell'interrupt esegue una funzione di salvataggio del contesto e invoca un gestore di eventi. Il gestore di eventi (event handler) analizza l'evento e cambia lo stato del processo, se qualcuno è coinvolto, per poi invocare lo scheduler a lungo termine, medio termine o breve termine, a seconda delle necessità.



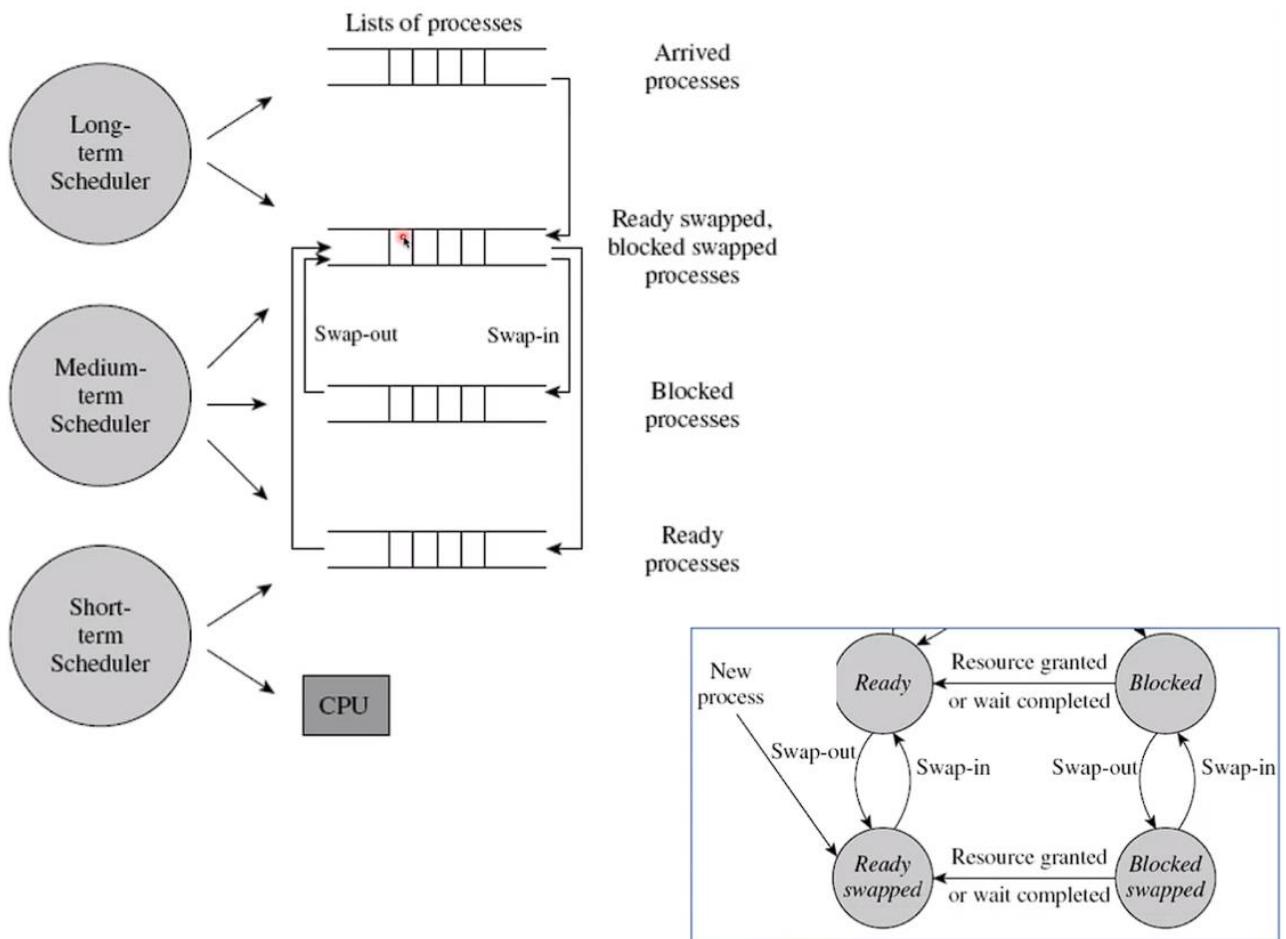
Scheduling nei sistemi Time-Sharing

Lo scheduler a **lungo termine** ammette un processo quando le risorse del kernel possono essere allocate. Il kernel copia il codice del processo nello spazio di swap e aggiunge il processo alla lista dei processi scaricati (swapped-out).

Lo scheduler a **medio termine** controlla lo swapping dei processi e decide quando spostare i processi tra le liste *ready-swapped* e *ready* e tra le liste *blocked-swapped* e *blocked*.

Ogni volta che la CPU è libera, lo scheduler a **breve termine** seleziona un processo dalla lista ready per l'esecuzione e il meccanismo di dispatching ne inizia o ripristina il funzionamento sulla CPU.

Un processo può passare tra gli scheduler a medio e a breve termine più volte come risultato di swapping.



nb: nei SO moderni, quando un processo viene creato viene direttamente ammesso nell'elenco dei processi schedulabili.

In tale soluzione invece i processi non partono da *ready* ma da *ready-swapped*.

STRUTTURE DATI e MECCANISMI DI SCHEDULING

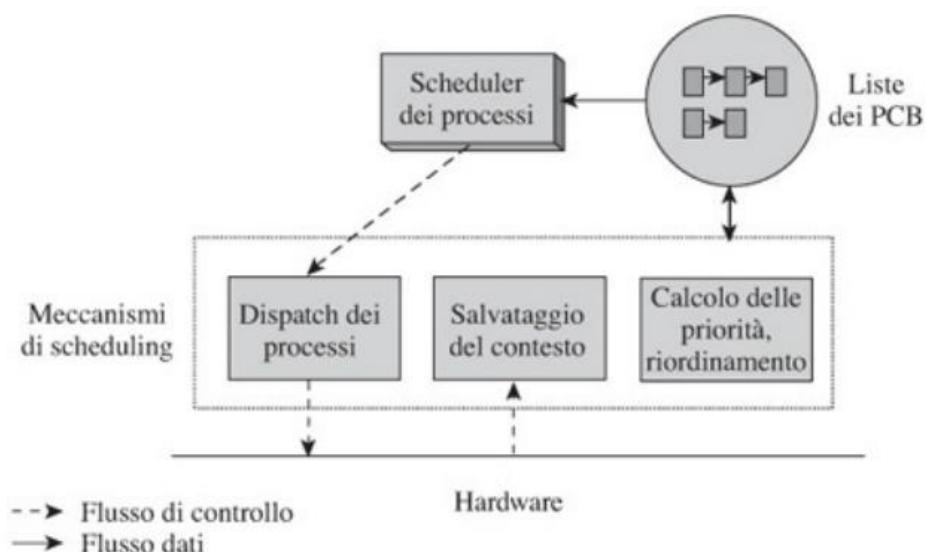
Lo scheduler utilizza un insieme di liste di PCB per selezionare i vari processi.

Lo scheduler preleva un processo da tale insieme e lo invia al *dispatcher*, il quale dovrà preoccuparsi di caricare il PSW e il GPR di tale processo nella CPU per ripristinare le operazioni del processo selezionato.

Il dispatcher interagirà da un lato con lo scheduler e dall'altro con l'HW in quanto opera con tali registri.

L'operazione di *salvataggio del contesto* fa parte delle operazioni dell'interrupt.

Se non ci sono processi *ready*, ovvero se l'insieme di liste PCB è vuoto, il kernel esegue un *loop idle* (non fa nulla) in attesa che dell'inserimento di una nuova lista di PCB.



nb: il loop idle provoca uno spreco di energie, per tal motivo in alcuni sistemi, in cui il risparmio energetico è più importante, la CPU può essere parzialmente o totalmente disattivata.

Esempi di **implementazione** sono:

- scheduling basato su priorità;
- scheduling Round-Robin con Time Slicing.

SCHEDULING BASATO SU PRIORITÀ

Per implementare uno scheduling basato su priorità, nella maggior parte dei SO, viene mantenuta una lista di processi *ready*, separata per ogni livello di priorità, e tutte organizzate come coda di PCB.

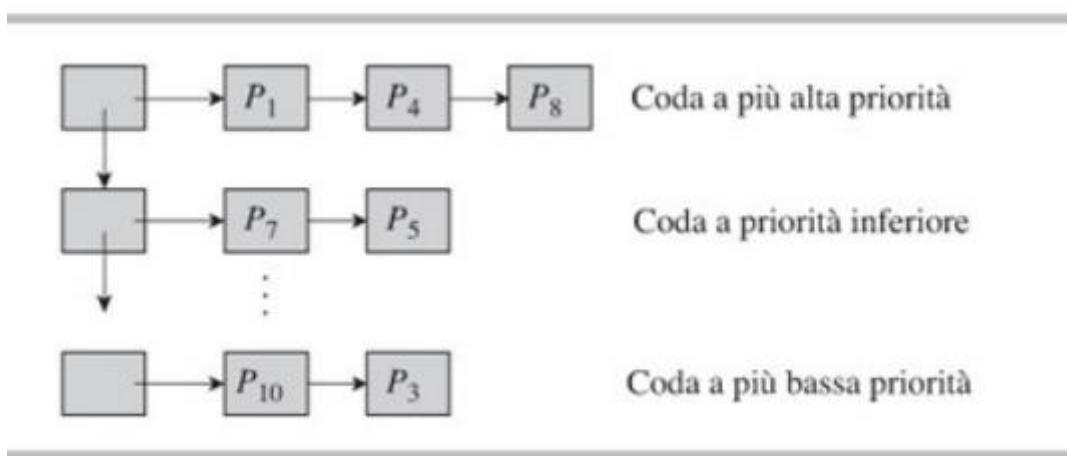
Tale insieme di code sono organizzate come una **lista** di puntatori a code di PCB, dove ogni elemento della lista prende il nome di **intestazione** ed ha due **puntatori**:

- uno, per l'appunto, che punta al primo elemento della coda;
- uno che punta all'elemento di intestazione successivo, che a sua volta punterà alla coda che ha il livello di priorità immediatamente inferiore rispetto alla coda precedente.

L'overhead sarà costituito dal numero di priorità distinte (più che dal numero di processi *ready*), in quanto per selezionare un processo si deve scorrere tali code.

Inoltre genera overhead anche la tecnica di *aging*, la quale deve essere implementata per impedire lo *starvation* dei processi a bassa priorità (infatti col passare del tempo un processo può essere spostato da una coda all'altra, il che richiede risorse).

Inoltre tale scheduling può causare l'*inversione di priorità*.



NB:

Aging: aumenta la priorità di un processo all'aumentare del tempo trascorso.

Inversione di priorità: è un problema che si verifica quando un processo ad alta priorità necessita della risorsa di un processo a più bassa priorità (che potrebbe richiedere molto tempo). Il processo ad alta priorità dovrà aspettare quello a più bassa priorità, quindi in effetti è come se avesse bassa priorità.

SCHEDULING ROUND-ROBIN CON TIME SLICING

Per implementare uno scheduling Round Robin con Time Slicing basta usare una singola lista di PCB dei processi *ready*, la quale è organizzata come una coda.

Lo scheduler rimuove il primo PCB dalla coda e schedula il processo corrispondente:

- Se il *time slice* scade, il PCB è messo alla fine della coda;
- Se il processo avvia un'operazione di I/O, il suo PCB è aggiunto alla fine della coda quando la sua operazione di I/O è completata.

In tale schema, col passare del tempo, si ha che un processo migra dal fondo della coda fino alla testa, dove il processo viene infine schedulato.

SCHEDULING MULTILIVELLO

Lo scheduling multivello è uno schema molto utilizzato nei SO moderni, e prevede di combinare lo *scheduling con priorità* con quello RR.

È detto *multilivello* in quanto si prevedono diverse code di priorità, dove ciascuna coda di ciascun livello di priorità viene gestita con una strategia di tipo RR.

L'idea è assegnare una maggior priorità ai **processi interattivi**, in quanto richiedono un tempo di risposta immediato, e minor priorità ai processi non interattivi.

In tal modo è anche possibile assegnare a ciascuna coda un *time slice* adeguato:

- nelle code a priorità maggiore il *time slice* è più breve così da garantire tempi di risposta più veloci;
- nelle code a priorità inferiore il *time slice* è più ampio così da evitare troppe commutazioni e garantire un più basso overhead.

Un processo in testa ad una coda è schedulato solo se le code per tutti i livelli più elevati di priorità sono vuote.

È uno scheduling con prelazione.

Le priorità sono *statiche*, cioè sono determinate a priori.

Ciò significa che se le priorità sono poste in modo errato il sistema ne risentirà, infatti non sarà possibile effettuare cambi di priorità né gestire eventuali *starvation*.

Per tal motivo è stato introdotto una *variante* di tale schema detto **Scheduling Adattivo Multilivello**, il quale è in grado di modificare le priorità delle code ed ha il compito di determinare ad ogni istante il livello di priorità "corretto" per ogni processo osservando il suo uso recente di CPU e I/O

SCHEDULING REAL-TIME

Lo scheduling real-time, rispetto a quelli ordinari, deve gestire inoltre **due vincoli di scheduling speciali** altrimenti ne risente il funzionamento dell'applicazione:

1. I processi nelle applicazioni sono processi **interattivi** che hanno delle loro *deadline*, ciò significa che una strategia di scheduling deve avere un modo per trasformare le deadline di un'applicazione in deadline appropriate per i processi.
2. I processi possono essere **periodici** (ripetono il loro funzionamento ogni TOT di tempo), ciò significa che una strategia di scheduling ne debba rispettare le *deadline*.

Precedenze di processo e Schedulazioni ammissibili

Le applicazioni real-time sono applicazioni in cui si ha un certo numero di processi, i quali interagiscono fra di loro garantendo certe *dipendenze*, chiamate **precedenze**.

Ciò significa che lo scheduler, quando seleziona un processo, deve tenere conto sia delle *deadline* che delle *precedenze*.

Terremo conto delle precedenze dei processi attraverso un **grafo**, e dove un processo P_i che precede un processo P_j sarà indicato come $P_i \rightarrow P_j$, ovvero P_j non può iniziare la sua attività finché P_i non termina la sua.

Una **grafo delle precedenze** dei processi (PPG) è un grafo orientato $G = (N, E)$ in cui i *nodi* sono rappresentati dai processi e gli *archi* sono rappresentati dalle dipendenze fra i processi (un arco $(P_i, P_j) \in E$ implica $P_i \rightarrow P_j$).

All'interno del grafo possiamo avere dei **cammini**, cioè una sequenza di esecuzioni di processi che va da P_i a P_k , nel quale c'è una relazione di precedenza fra ogni coppia di nodi.

In tal caso P_k è detto *discendente* di P_i , e può essere discendente *diretto* o *indiretto*:

- Se P_k è un discendente *diretto* di P_i allora si ha direttamente un arco fra P_i e P_k , e la relazione è indicata con “ \rightarrow ”,
- Se P_k è un discendente *indiretto* di P_i allora si hanno processi intermedi fra P_i e P_k , e la relazione è indicata con “ \rightarrow^* ”.

Tale grafo verrà utilizzato per verificare se i vincoli temporali sono soddisfatti, e il modo con cui posso soddisfare i vincoli dipende dal tipo di sistema:

- **sistema real-time hard**: è garantito assolutamente il rispetto delle deadline;
- **sistema real-time soft**: non è garantito il rispetto delle deadline, ergo i vincoli temporali sono soddisfatti *probabilisticamente*.

Definito un grafo delle precedenze diremo che,

Una **schedulazione ammissibile** è una sequenza di decisioni di scheduling che permette ai processi di un'applicazione di operare in accordo con le rispettive precedenze e di soddisfare i requisiti (vincoli) dell'applicazione.

Ergo, lo **scheduling real-time** è orientato sull'implementazione di una **schedulazione ammissibile**, se ne esiste una.

Lo scheduling real-time può essere realizzato attraverso diverse **strategie**:

Approccio	Descrizione
Scheduling statico	Uno schedule viene preparato <i>prima</i> che l'esecuzione dell'applicazione real-time cominci, tenendo conto delle interazioni tra processi, le periodicità, i vincoli sulle risorse e le scadenze.
Scheduling basato su priorità	L'applicazione real-time viene analizzata per attribuire appropriate <i>priorità</i> ai suoi processi. Durante l'esecuzione dell'applicazione viene adottato lo scheduling convenzionale basato su priorità.
Scheduling dinamico	Lo scheduling è eseguito quando <i>proviene</i> una richiesta di creazione di un processo. La creazione di un processo avviene solo se il requisito di risposta del processo può essere soddisfatto in maniera garantita.

Per i sistemi *real-time soft* è possibile utilizzare un'ulteriore politica che è una *variante* dello scheduling dinamico, detta **scheduling ottimista**: essa ammette tutti i processi, ma può perdersi qualche deadline.

L'**ammissibilità delle schedulazione** può essere verificata con:

- Scheduling con Deadline;
- Scheduling per Processi Periodici.

SCHEDULING CON DEADLINE

Prima di approfondirne il funzionamento, concentriamoci sulle *deadline*.

Possono essere specificati due tipi di deadline:

- *deadline di inizio*: l'ultimo istante di tempo entro cui le operazioni del processo devono iniziare;
- *deadline di completamento*: istante in cui le operazioni del processo devono terminare (è quella che consideriamo).

La **stima delle deadline** è fatta considerando le precedenze dei processi e lavorando all'indietro dal requisito di risposta dell'applicazione, e possiamo ottenerla come:

$$D_i = D_{\text{application}} - \sum_{k \in \text{descendant}(i)} x_k$$

Dove $D_{\text{application}}$ è la scadenza dell'applicazione e x_k è il tempo di servizio del processo P_k .

Cioè, posso calcolare la deadline del processo i -esimo D_i considerando la deadline dell'intera applicazione $D_{\text{application}}$ e togliendoci i tempi di servizio di ciascun processo che è descendente di P_i .

Se ripeto questo per ogni processo, avrò la deadline di ogni processo.

nb: per semplicità, tale stima non tiene in considerazione né la prelazione dei processi né le operazioni di I/O.

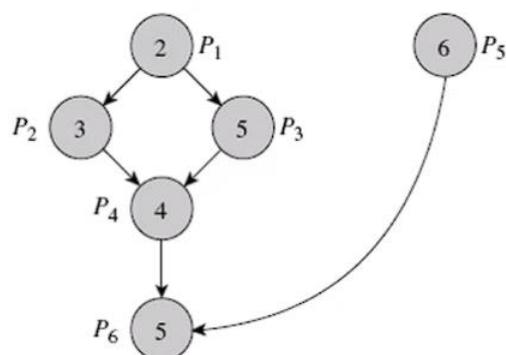
Esempio:

Si tratta di un grafo di 6 processi, dove possiamo distinguere diverse dipendenze. Il totale dei tempi di servizio dei processi è 25 secondi, quindi $D_{\text{application}} = 25$.

Lo scopo sarà quello di calcolare le deadline di ogni processo, ovvero D_i per $i = 1:6$, e dove D_i è ottenuto come il totale dei tempi di servizio meno la somma dei tempi di servizio dei processi che dipendono da P_i .

Ad esempio, $D_1 = 25 - (3+5+4+5) = 8$.

Se calcoliamo ogni deadline, avremo che:



Process	P_1	P_2	P_3	P_4	P_5	P_6
Deadline	8	16	16	20	20	25

Nella pratica, la determinazione delle deadline è più complesso in quanto si deve tener conto di molti altri vincoli (prelazioni e overlap con operazioni I/O), tuttavia sono comunque calcolabili.

Calcolate le deadline, utilizziamo lo scheduling **Earliest Deadline First** (EDF) per selezionare sempre il processo con la deadline più prossima.

- Sia **seq** la sequenza in cui i processi sono elaborati;
- Sia **pos(P_i)** la posizione di P_i nella sequenza delle decisioni di scheduling;

Lo sforamento della deadline di P_i , ovvero D_i , non avviene se:

$$\sum_{k: pos(P_k) \leq pos(P_i)} x_k \leq D_i$$

cioè, se la somma dei tempi di servizio di tutti i processi che lo precedono (incluso lui) è inferiore alla sua deadline.

Se tale condizione è **rispettata** allora **esiste schedulazione ammissibile**.

Una strategia di questo è *semplice*, in quanto è facile da implementare, ed è *senza prelazione*. Inoltre, si tratta di una *buona politica* per lo scheduling statico (o anche per quello dinamico ma solo per i sistemi real-time soft).

Il difetto di tale algoritmo è che, in presenza di schedulazioni non ammissibili, lo scheduling non è predicibile.

Ovvero supponiamo di avere due applicazioni indipendenti, e che tutti i processi devono essere completati entro un certo tempo, allora non esiste una schedulazione ammissibile in quanto non saremo in grado di prevedere quale sequenza sarà scelta dall'EDF.

SCHEDULING PER PROCESSI PERIODICI

I processi periodici sono processi che possono ripetersi nel tempo ad intervalli regolari, intervalli di cui dovrò tener considerazione.

Per semplicità, supponiamo che non ci siano operazioni di I/O e che i processi siano indipendenti (non c'è nessun grafo da considerare).

Si considera la **frazione del tempo di CPU usato** P_i come il rapporto fra il tempo di servizio e il periodo: $P_i = x_i / T_i$

In generale, l'insieme dei processi periodici $P_1 \dots P_n$ che non eseguono I/O può essere servito da un sistema *real-time hard* che ha un overhead trascurabile se:

$$\sum_{i=1 \dots n} \frac{x_i}{T_i} \leq 1$$

cioè, se la somma di tali rapporti è inferiore ad 1 allora **esiste schedulazione ammissibile**.

esempio:

Dati questi tre processi, avremo che:

$$P_1 = 10/3$$

$$P_2 = 15/5$$

$$P_3 = 30/9$$

Processo	P_1	P_2	P_3
Periodo (ms)	10	15	30
Tempo di servizio (ms)	3	5	9

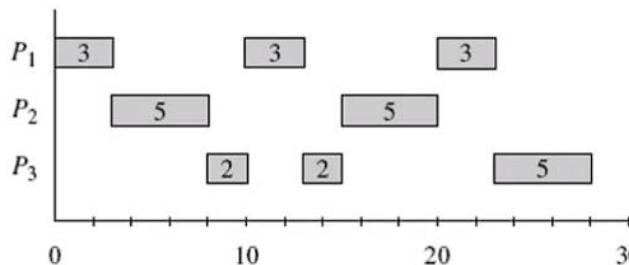
La cui somma è 0.93, che è < 1 ; quindi esiste schedulazione ammissibile.

Verificata l'ammissibilità, determiniamo la priorità dei processi attraverso l'algoritmo **Scheduling Rate Monotonic** (SRM), con il quale i processi con il periodo più breve avranno priorità maggiore.

La priorità del processo i -simo verrà quindi calcolata come il reciproco del tasso i -simo.

$$\text{Tasso di } P_i = 1/T_i$$

Calcolate le priorità, utilizzerò semplicemente uno scheduling con priorità.



Processo	P_1	P_2	P_3
Periodo (ms)	10	15	30
Tempo di servizio (ms)	3	5	9

Notare tuttavia che l'SRM non garantisce uno scheduling ammissibile in tutte le situazioni. Infatti può capitare che si sfiori la deadline di alcuni processi e che quindi non si ha scheduling ammissibile.

In generale, diremo che l'SRM non rispetti alcuni deadline di processi se non viene rispettata la seguente relazione:

$$\sum_{i=1}^m \frac{x_i}{T_i} \leq m \left(2^{\frac{1}{m}} - 1 \right) \rightarrow 0.69 \text{ per } m \rightarrow \infty$$

cioè, se la somma del tempo di utilizzo di ciascun processo è inferiore a 0.69.

Infatti la relazione a destra tende a 0.69 per m che tende ad infinito dove m indica il numero dei processi.

Ovvero, se l'applicazione ha un elevato numero di processi, può non essere in grado di raggiungere più del **69%** di utilizzo di CPU se deve rispettare la deadline dei processi.

Per ovviare a tale problema si dovrebbe utilizzare una variante dell'SRM, lo *scheduling guidato da deadline*, che permette di raggiungere il 100% di utilizzo di CPU (ma si ha maggior overhead).

LEZIONE 15 – DEADLOCK (1)

Un deadlock si ha quando in un insieme di processi esistono processi dell'insieme che attendono un evento che deve essere generato da un altro processo dell'insieme. Ogni processo è dunque in attesa di un evento che potrebbe non verificarsi mai.

Formalmente, un deadlock è una situazione che coinvolge un insieme di processi D in cui ogni processo P_i in D soddisfa due condizioni:

1. il processo P_i è bloccato da qualche evento e_j .
2. l'evento e_j può essere causato solo da azioni prodotte da un altro o più processi in D .

Gli **stalli** possono essere associati alle **risorse**, e in tal caso è una preoccupazione del SO, oppure possono verificarsi nella **sincronizzazione** e nella **comunicazione** di messaggi, e in tal caso è una preoccupazione dell'utente.

DEADLOCK NELL'ALLOCAZIONE DI RISORSE

Un SO può contenere molte risorse di un certo tipo:

- un'**unità di risorsa** si riferisce ad una risorsa di tipo specifico;
- una **classe di risorse** si riferisce all'insieme di tutte le unità di risorsa di un tipo.

Indichiamo con R_i una classe di risorse e con r_j un'unità di risorsa di una classe.

L'allocazione delle risorse in un sistema implica tre tipi di **eventi**:

Evento	Descrizione
Richiesta	Un processo richiede una risorsa tramite una chiamata di sistema. Se la risorsa è libera, il kernel la alloca al processo immediatamente; altrimenti, cambia lo stato del processo a <i>blocked</i> .
Allocazione	Il processo diventa <i>holder</i> della risorsa a esso allocata. Le informazioni sullo stato della risorsa vengono aggiornate e lo stato del processo diventa <i>ready</i> .
Rilascio	Un processo rilascia una risorsa tramite una chiamata di sistema. Se vari processi sono bloccati sull'evento allocazione della risorsa, il kernel usa alcune regole, come l'allocazione FCFS, per decidere a quale processo allocare la risorsa.

Condizioni per il Deadlock di risorsa

Le condizioni sono 4, e devono verificarsi simultaneamente affinché si verifichi un deadlock:

Condizione	Descrizione
Risorse non condivisibili o mutua esclusione	Risorse che non possono essere condivise; un processo necessita di accesso esclusivo alla risorsa.
Assenza di prelazione	Una risorsa non può essere prelevata di autorità da un processo e allocata a un altro.
Possesso e attesa	Un processo continua a tenere la risorsa allocata in attesa di altre risorse.
Attesa circolare	Esiste nel sistema una catena circolare di condizioni possesso e attesa; per esempio, il processo P_i aspetta P_j , P_j aspetta P_k e P_k aspetta P_i .

L'*attesa circolare* non può verificarsi se le 3 condizioni precedenti non sono verificate, per tal motivo devono tutte verificarsi simultaneamente.

Inoltre, è essenziale anche un'**altra condizione** per il deadlock;

Nessun annullamento di richieste di risorse: un processo bloccato su una richiesta di risorsa non può annullarla.

Si tratta di una condizione verificata *implicitamente* in quanto se non valesse tale condizione non valrebbero nemmeno le 4 fondamentali.

Stato di allocazione delle risorse

Lo stato di allocazione delle risorse *riguarda* le informazioni sulle risorse allocate ai processi e sulle richieste pendenti di risorse, usato per determinare se un insieme di processi è in deadlock.

Per rappresentare tale stato sono usati due tipi di **modelli**:

- **Grafo**: un processo può richiedere e usare esattamente un'unità di risorsa di ogni classe di risorse.
- **Matrice**: un processo può richiedere un qualsiasi numero di unità di una classe di risorse.

Si noti come il modello a Matrice sia più generale, in quanto può essere usato non solo per classi di risorse con più di un'istanza ma anche quando un processo richiede più risorse per ciascuna classe, mentre nel modello a Grafo tale rappresentazione non è possibile.

Per quanto riguarda i grafi, distinguiamo: RRAG e WFG.

Grafi RRAG e WFG

Un **RRAG** (*Resource and Request Allocation Graph*) è un grafo in cui:

- esistono **due tipi di nodi**:
 1. *processo*, rappresentato da un cerchio;
 2. *classe di risorse*, rappresentato da un rettangolo. Ciascun punto in un rettangolo è un'unità di risorsa.
- esistono **due tipi di archi**:
 1. *arco di allocazione di risorsa*: un arco che va **da una classe di risorse ad un processo**; es. (R_k, P_j)
 2. *arco di richiesta pendente di risorsa*: un arco che va **da un processo ad una classe di risorse** (il processo è bloccato sulla richiesta dell'unità di risorsa di quella classe). es. (P_i, R_k)

Un arco di allocazione (R_k, P_j) è cancellato quando il processo P_j rilascia un'unità di risorsa della classe R_k .

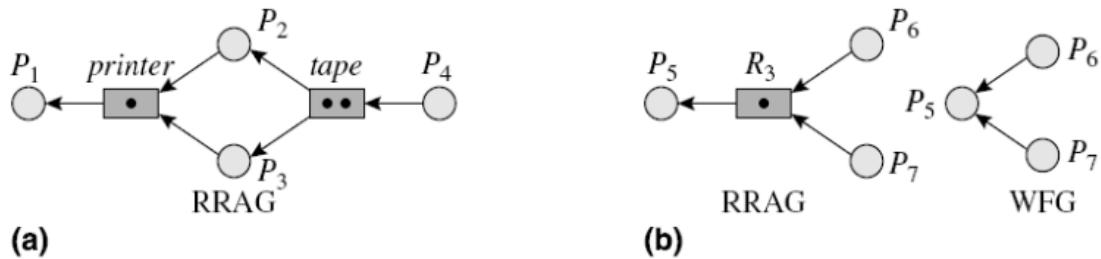
Un arco di richiesta (P_i, R_k) è cancellato ed è aggiunto un arco di allocazione (R_k, P_i) quando è concessa una richiesta in attesa del processo P_i per un'unità di una classe R_k .

Un **WFG** (*Wait For Graph*) è un grafo, più semplice, che può essere usato per descrivere lo stato delle risorse di un **sistema in cui ogni classe di risorse contiene solo un'unità di risorsa**.

Esiste **un solo tipo di nodo**, che rappresenta i *processi*, ed un **solo tipo di arco**, che rappresenta una *relazione wait-for* tra processi.

Un arco wait-for (P_i, P_j) indica che: il processo P_j ha ottenuto una singola risorsa da una classe e P_i è in attesa che P_j rilasci tale risorsa.

RRAG vs WFG



(a) Grafo di richiesta e allocazione di risorse (RRAG);

(b) Equivalenza di RRAG e WFG quando ogni classe di risorse contiene una sola unità.

(a) P_1 ha avuto allocato un'unità della classe stampante, e i processi P_2 e P_3 sono bloccati in attesa che si liberi tale unità. A loro volta, i processi P_2 e P_3 hanno avuto allocato un'unità della classe nastro, e il processo P_4 è in attesa che si liberi almeno un'unità. Visto che c'è almeno una classe di risorsa con più unità, non si può usare il WFG.

(b) P_5 ha avuto allocato un'unità da una certa classe, e i processi P_6 e P_7 sono bloccati in attesa che si libri tale unità. Visto che è presente una sola unità per ogni classe, si può usare il WFG.

Cammini nei WFG e RRAG

Un cammino in un grafo è una sequenza di archi tali che il nodo destinazione di un arco è il nodo sorgente dell'arco seguente.

Consideriamo un cammino in un **RRAG**, dove in generale si ha che ad un Processo P_n viene allocata una risorsa da R_{n-1} , la quale è attesa da un processo P_{n-1} che a sua volta viene allocata una risorsa da R_{n-2} e così via...

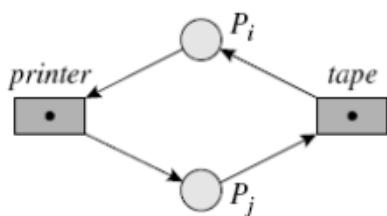
Il cammino sarebbe dunque: $P_1 - R_1 - P_2 - R_2 \dots P_{n-1} - R_{n-1} - P_n$.

Nel **WFG**, lo stesso cammino sarebbe: $P_1 - P_2 \dots P_{n-1} - P_n$.

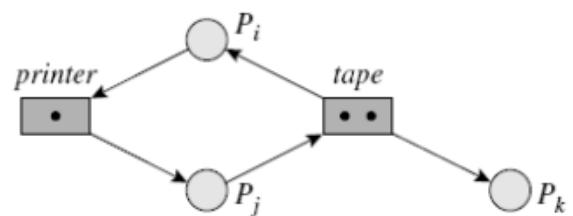
Supponiamo che ogni classe di risorsa contenga un'unica unità, allora entrambi i cammini visualizzati dai grafi sono **privi di deadlock**.

Questo perché sebbene i processi da P_1 a P_{n-1} siano bloccati, il processo P_n non è bloccato e terminata la sua esecuzione rilascerà la risorsa al processo P_{n-1} , e così via.

Infatti, non può esistere un deadlock se un RRAG o un WFG **NON contiene un ciclo**. Tuttavia un ciclo in un RRAG non implica necessariamente che un deadlock se una classe di risorse ha unità multiple.



(a)



(b)

(a) P_i ha avuto allocato il nastro e richiede la stampante, tuttavia la stampante era stata allocata a P_j che però a sua volta richiede il nastro.

Si è formato un ciclo in cui una delle due risorse può essere rilasciata solo da uno dei due processi, che essendo bloccati sono in stallo.

(b) situazione simile ad a, però una delle due unità del nastro è allocata a P_k . Ciò significa che nel momento che P_k termina, la sua unità del nastro può essere allocata a P_j .

Matrice

Lo stato di allocazione delle risorse è rappresentato da due matrici: *risorse allocate* e *risorse richieste*.

Se un sistema ha n processi e r classi di risorse, ciascuna di tali matrici ha dimensione $n \times r$.

Spesso vengono utilizzate due ulteriori strutture dati, che tengono in considerazione le *risorse totali* e le *risorse libere* per ogni classe di risorsa.

ES.

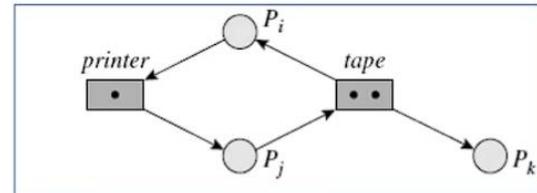
	Printer	Tape
P_i	0	1
P_j	1	0
P_k	0	1

Allocated resources

	Printer	Tape
P_i	1	0
P_j	0	1
P_k	0	0

Requested resources

Total resources	Printer	Tape
Free resources	1	2
	0	0



GESTIONE DEI DEADLOCK

Possiamo distinguere **3 strategie fondamentali** per gestire i deadlock, dove ognuna può essere preferita ad un'altra in base alla situazione:

Approccio	Descrizione
Rilevamento e risoluzione dei deadlock	Il kernel analizza lo stato della risorsa per controllare se esiste un deadlock. Se è così, interrompe alcuni processi e assegna le risorse allocate ad altri processi affinché il deadlock cessi di esistere.
Prevenzione dei deadlock	Il kernel usa una politica di allocazione delle risorse che assicura che le quattro condizioni per i deadlock di risorsa menzionate in Tabella 8.2 non si verifichino simultaneamente. Questo rende impossibili i deadlock.
Evitare i deadlock	Il kernel analizza lo stato di allocazione per determinare se l'accettazione di una richiesta di risorsa può determinare un deadlock. Vengono accettate solo richieste che non conducono a deadlock, le altre vengono tenute in attesa finché possono essere accettate. In questo modo, non si verificano deadlock.

nb: l'approccio 3 è diverso dall'1 perché nel 3 il kernel analizza lo stato di allocazione prima di concedere le risorse ad un processo, e nel caso si formi un deadlock tali risorse non vengono proprio assegnate al processo.

RILEVAMENTO E RISOLUZIONE DEI DEADLOCK

Un processo bloccato **non è coinvolto** in un deadlock se la richiesta su cui è bloccato può essere soddisfatta con la sequenza di eventi:
completamento processo – rilascio risorsa – allocazione risorsa.

Cioè, un processo è bloccato da un processo che però è in esecuzione, e al momento del completamento rilascerà la risorsa che verrà poi riallocata.

Per **verificare se ci sono deadlock** possiamo utilizzare il modello a grafo o quello a matrice:

- **grafo:** se ciascuna classe di risorsa in un sistema contiene una singola unità, il controllo avviene verificando la presenza di cicli nel RRAG o WFG.
- **matrice:** si cerca di costruire una sequenza ammissibile di eventi dove tutti i processi bloccati possono ottenere le risorse che hanno richiesto.
Se il tentativo va a buon fine allora non c'è deadlock, altrimenti sì.

Notare che il modello su grafi non è applicabile se le classi di risorse hanno più unità, e quindi sarebbero necessari algoritmi più complessi per i RRAG.

Invece il controllo a matrice è applicabile in tutte le situazione (*preferito*).

Esempio:

Ai fini dell'individuazione del deadlock, a noi interessa solo quando il processo è in esecuzione (*running*) o bloccato. Gli altri stati non ci interessano.

Supponiamo che lo stato di un sistema contenga 10 unità di una classe di risorse R_1 e tre processi P_1, P_2 e P_3 .

	R_1	R_1	Total resources	R_1
P_1	4	6	10	
P_2	4	2		
P_3	2	0		
Allocated resources		Requested resources	Free resources	0

Dallo schema si nota come P_1 abbia richiesto in totale 10 risorse, di cui solo 4 gli sono stati *allocate* (le rimanenti 6 sono solo state *richieste*). P_2 ha richiesto 6 risorse, di cui solo 4 gli sono state allocate. P_3 ha richiesto 2 risorse, che le sono state allocate tutte.

Il processo P_3 è nello stato *running*, ciò significa che alla sua terminazione le risorse verranno allocate ad un altro processo. Se tale processo è P_2 , significa che tale processo ora ha tutte e 6 le risorse di cui aveva bisogno e può andare nello stato *running*.

Tutti i processi in questo modo possono completare, e non esiste dunque alcun deadlock.

Algoritmo di individuazione del Deadlock

Algoritmo 8.1 Individuazione dei deadlock

Input

<i>n</i>	:	numero di processi;
<i>r</i>	:	numero di classi di risorsa;
<i>Blocked</i>	:	insieme di processi;
<i>Running</i>	:	insieme di processi;
<i>Risorse.Libere</i>	:	array [1..r] of integer;
<i>Risorse.Allocate</i>	:	array [1..n, 1..r] of integer;
<i>Risorse.Richieste</i>	:	array [1..n, 1..r] of integer;

Strutture dati

<i>Exit</i>	:	insieme di processi;
-------------	---	-----------------------------

1. **repeat until** l'insieme *Running* è vuoto
 - a. seleziona un processo P_i dall'insieme *Running*;
 - b. cancella P_i dall'insieme *Running* e aggiungilo all'insieme *Exit*;
 - c. **for** $k = 1..r$
 $Risorse.Libere[k] = Risorse.Libere[k] + Risorse.allocate[i, k];$
 - d. **while** l'insieme *Blocked* contiene un processo P_j tale che
 - i. **for** $k = 1..r$
 $Risorse.Libere[k] = Risorse.Libere[k] - Risorse.richieste[j, k];$
 $Risorse.allocate[j, k] = Risorse.allocate[j, k]$
 $+ Risorse.richieste[j, k];$
 - ii. Rimuovi P_j dall'insieme *Blocked* e aggiungilo all'insieme *Running*;
2. **if** l'insieme *Blocked* è non vuoto **then**
imposta i processi nell'insieme *Blocked* come processi in deadlock.

Tale algoritmo ripercorre più o meno l'esempio sopra.

Esso eseguirà un ciclo in cui andrà a controllare tutti i processi in running finché l'insieme *Running* non è vuoto.

Ad ogni iterazione selezionerà un processo *running*, lo rimuove dall'insieme *Running* e invece lo aggiunge all'insieme *Exit*, e ne libera le risorse associate.

Una volta liberate, verrà osservato l'insieme dei processi *Blocked*, nel quale controllerà che per ogni processo bloccato P_i , le risorse che richiede siano inferiori alle risorse disponibili, e se così fosse gli andrà ad associare le risorse.

A tal punto eliminerò P_i dall'insieme *Blocked* e lo aggiungerò a quello *Running*.

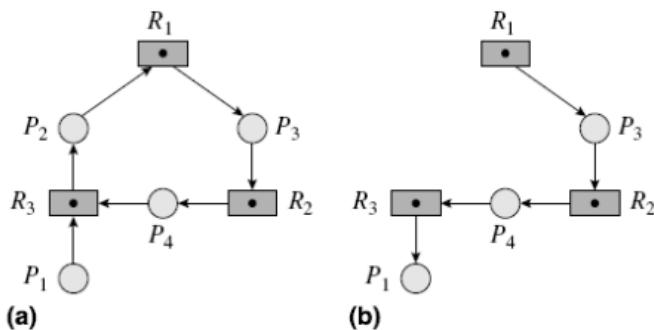
Terminato il ciclo, se l'insieme *Blocked* non è vuoto significa che c'è qualche processo in deadlock, altrimenti nessun processo è in deadlock.

Risoluzione Deadlock

La risoluzione del deadlock per un insieme D di processi in deadlock consiste nello spezzare il deadlock per assicurare il progresso per alcuni processi in D.

Si ottiene **forzando** la terminazione di uno o più processi in D. Ogni processo terminato è chiamato *vittima*, e le sue risorse saranno poi allocate ad altri processi.

La scelta della vittima può essere effettuata su diversi criteri.



Risoluzione deadlock. (a) un deadlock; (b) stato allocazione risorse dopo la risoluzione del deadlock

PREVENZIONE DEADLOCK

Per prevenire un deadlock dobbiamo tenere in considerazione le 4 condizioni che portano alla formazione di un deadlock.

Le **strategie** di prevenzione mirano a risolvere una delle 4 condizioni, in quanto se una non si verifica non si possono verificare le altre.

a) **risorse non condivisibili o mutua esclusione**: rendo le risorse condivisibili, quindi non c'è attesa di un processo rispetto ad un altro;

b) **assenza di prelazione**: rendo le risorse prelazionabili, cioè se un processo P_i è in attesa perché una risorsa è assegnata ad un altro processo, prelaziono la risorsa dal processo che la possiede e la assegnano al processo P_i ;

c) **possesso e attesa**: si "viola" la condizione, e si può fare in due modi:

1. si fa in modo che il processo P_i non possa bloccarsi nella richiesta di una risorsa già impegnata;
2. si fa in modo che il processo P_i , bloccato, liberi le altre risorse che manteneva.

ovvero, non si può avere nel grafo dei percorsi con più di un processo.

Quindi non posso avere percorsi circolari, e di conseguenza deadlock.

d) **attesa circolare**: si previene l'attesa circolare introducendo un *vincolo di validità*, il quale è un valore booleano che sarà vero se la richiesta del processo non porta ad una attesa circolare, altrimenti sarà falso.

Se il vincolo è vero vengono attribuite immediatamente le risorse al processo, altrimenti esso viene bloccato finché tali risorse non vengono liberate.

Approccio	Rappresentazione	
	Senza questo approccio	Con questo approccio
(a) Rendere le risorse condivise → Nessuna attesa Il processo P_i non si blocca sulla risorsa R_j .		
(b) Rendere le risorse prelazionabili → Nessuna attesa circolare La risorsa R_j viene prelazionata ed allocata a P_i .		
(c) Prevenire hold-and-wait → Nessun percorso in RRAG con numero di processi > 1 Al processo P_i (1) non è consentito bloccarsi sulla risorsa R_j , o (2) non è consentito mantenere R_k mentre richiede R_j .		
(d) Prevenire attese circolari Al processo P_j non è permesso richiedere la risorsa R_i .		

Prevenzione deadlock: Allocazione simultanea

È la più semplice strategia di prevenzione deadlock. Un processo deve chiedere **tutte le risorse di cui necessita** con un'unica richiesta, le quali saranno allocate tutte insieme dal kernel.

Così facendo un processo blocked non possiede alcuna risorsa, e quindi non si verificherà mai la condizione di “possesso e attesa”.

Sebbene sia una strategia interessante per piccoli SO, porta un **degrado notevole delle prestazioni** in quanto un processo mantiene le risorse per un tempo più lungo del necessario. Infatti alcune di queste risorse potrebbero non essere usate immediatamente, non consentendo ai processi che hanno bisogno di quelle risorse di sbloccarsi.

Prevenzione deadlock: Ranking delle risorse

Ad ogni classe di risorse è associato un *rank di risorsa*. Alla richiesta di risorsa, il kernel applica un **vincolo di validità** per decidere se soddisfare la richiesta.

Tale vincolo consiste nel verificare che il rank della risorsa richiesta sia maggiore della risorsa con rank più elevato correntemente allocata al processo:

- **TRUE**: la risorsa è allocata al processo, se disponibile, altrimenti è bloccato in attesa che la risorsa sia rilasciata;
- **FALSE**: la richiesta è rifiutata ed il processo richiedente è interrotto.

Tale vincolo fondamentale stabilisce un ordine con cui richiedere le risorse.

Così facendo non si verificherà mai la condizione di “attesa circolare”.

Tale strategia **funziona al meglio** quando tutti i processi richiedono le rispettive. Nel caso peggiore, la strategia può degenere nella strategia di “allocazione simultanea”.

LEZIONE 16 – DEADLOCK (2)

EVITARE I DEADLOCK

Una richiesta di risorse, per un dato processo, verrà soddisfatta solo se si può stabilire che concedendo le risorse a quel processo non si verificherà un deadlock né nell'immediato e né nel futuro.

Sebbene sia semplice individuare un deadlock nell'immediato, è più complicato individuare deadlock nel futuro in quanto il kernel non è a conoscenza del comportamento futuro dei processi.

Si utilizza un **approccio conservativo**, cioè ogni processo è tenuto a dichiarare il numero massimo di risorse che può richiedere, per ciascuna classe, durante la sua esecuzione. Il kernel si occuperà di fornire di volta in volta le risorse necessarie al processo, e non è detto che il processo usi tutte le risorse che ha dichiarato.

È detto “conservativo” proprio perché nella simulazione deve tener conto del caso peggiore, e costringe quindi alcuni processi ad aspettare più di quanto dovrebbero effettivamente.

Algoritmo del banchiere

L'analogia è che i banchieri ammettono dei prestiti che collettivamente superano i fondi della banca e quindi concedono il prestito di ciascun mutuatario a rate.

Per ogni nuova richiesta di un processo, il sistema valuta se la richiesta, attraverso una simulazione, da origine ad un deadlock. In tal caso le risorse non verranno concesse. L'algoritmo del banchiere verifica se concedendo delle risorse ad un dato processo, si possa transire da uno **stato sicuro** all'altro.

notazione:

Notazione	Significato
$Risorse_richieste_{j,k}$	Numero di unità della classe di risorsa R_k attualmente richieste dal processo P_j
$Risorse_massime_{j,k}$	Massimo numero di unità della classe di risorsa R_k di cui può aver bisogno il processo P_j
$Risorse_allocate_{j,k}$	Numero di unità della classe di risorsa R_k allocate al processo P_j
$Risorse_totali_allocate_k$	Numero totale di unità allocate della classe di risorsa R_k , ossia $\sum_j Risorse_allocate_{j,k}$
$Risorse_totali_k$	Numero totale di unità della classe di risorsa R_k appartenenti al sistema

stato di allocazione sicuro: è uno stato di allocazione in cui è possibile costruire una sequenza di eventi *completamento processo – rilascio risorsa – allocazione risorsa* con cui ogni processo P_j nel sistema può ottenere $Risorse_massime_{j,k}$ risorse per ogni classe di risorsa R_k e completare le proprie operazioni.

Schema dell'approccio

1. Quando un processo fa una richiesta, fa una *proiezione dello stato di allocazione*, ovvero verifica come sarebbe lo stato se la richiesta fosse soddisfatta;
2. Se tale proiezione è sicura, concede le risorse e aggiorna *Risorse_allocate* e *Risorse_totali_allocate*; altrimenti, mantiene la richiesta pendente:
 - a. la sicurezza è verificata con una simulazione (cerca di costruire una sequenza **completamento-rilascio-allocazione** con tutti i processi che possono terminare)
 - b. è assunto che un processo completa le sue operazioni solo se può prendere il massimo richiesto di ciascuna risorsa simultaneamente, ovvero, **per tutti i k**.

$$Risorse_{totali_k} - Risorse_{totali_allocate_k} \geq Risorse_{massime_{l,k}} - Risorse_{allocate_{l,k}}$$

3. Quando un processo rilascia una qualsiasi risorsa o termina le operazioni, esamina le richieste pendenti e alloca quelle che pongono il sistema in un nuovo stato sicuro.

Esempio (con una sola classe di risorse)

Un sistema contiene 10 unità di risorse di una singola classe R_k .

Il requisito di risorse massime dei tre processi è rispettivamente 8, 7, 5 e l'allocazione corrente è 3, 1, 3:

	P_1	P_2	P_3		Total alloc	7
	8	3	1		Total resources	10
Max need	8	7	5			
Allocated resources	3	1	3			
Requested resources						

- P_1 fa una richiesta di risorsa: dobbiamo verificare che non possano verificarsi deadlock, cioè dobbiamo verificare la formula vista sopra.
Si può applicare, in quanto anche se allochiamo la risorsa, osserviamo che il processo P_3 può terminare la sua esecuzione (nel caso richiedesse tutte le sue risorse) e quindi libera delle risorse che possono essere usate per completare gli altri processi.
Notare come quello che abbiamo fatto è stato semplicemente trovare una sequenza **completamento-rilascio-allocazione**.

Algoritmo 8.2 Algoritmo del banchiere

Input

<i>n</i>	: numero di processi;
<i>r</i>	: numero di classi di risorse;
<i>Blocked</i>	: insieme di processi;
<i>Running</i>	: insieme di processi;
<i>P_{processo_richiedente}</i>	: Processo che effettua la nuova richiesta di risorsa;
<i>Risorse_maxime</i>	: array [1..n, 1..r] di integer;
<i>Risorse_allocate</i>	: array [1..n, 1..r] di integer;
<i>Risorse_richieste</i>	: array [1..n, 1..r] di integer;
<i>Risorse_totali_allocate</i>	: array [1..r] di integer;
<i>Risorse_totali</i>	: array [1..r] di integer;

Strutture dati

<i>Active</i>	: insieme di processi;
<i>fattibile</i>	: boolean;
<i>Nuova_richiesta</i>	: array [1..r] di integer;
<i>Allocazione_simulata</i>	: array [1..n, 1..r] di integer;
<i>Risorse_totali_allocate_simulate</i>	: array [1..r] di integer;

1. $Active = Running \cup Blocked;$
for $k = 1..r$
 $Nuova_richiesta[k] = Risorse_richieste[processo_richiedente, k];$
2. $Allocazione_simulata := Risorse_allocate;$
for $k = 1..r$ /* Calcolare lo stato di allocazione proiettato*/
 $Allocazione_simulata[processo_richiedente, k] :=$
 $Allocazione_simulata[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate_simulate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$
3. $fattibile = true;$
for $k = 1..r$ /* Controllare se lo stato di allocazione proiettato è fattibile */
 $\text{if } Risorse_totali[k] < Risorse_totali_allocate_simulate[k] \text{ then } fattibile = false;$
4. **if** $fattibile = true$
then /* Controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro */
while l'insieme *Active* contiene un processo P_j tale che
Per ogni k , $Risorse_totali[k] - Risorse_totali_allocate_simulate[k]$
 $\geq Risorse_massime[l, k] - Allocazione_simulata[l, k]$
Rimuovi P_j da *Active*;
for $k = 1..r$
 $Risorse_totali_allocate_simulate[k] :=$
 $Risorse_totali_allocate_simulate[k] - Allocazione_simulata[l, k];$
5. **if** l'insieme *Active* è vuoto
then /* Lo stato di allocazione proiettato è uno stato di allocazione sicuro */
for $k = 1..r$ /* Cancellare la lista dalle richieste in attesa */
 $Risorse_richieste[processo_richiedente, k] := 0;$
for $k = 1..r$ /* Accettare la richiesta */
 $Risorse_allocate[processo_richiedente, k] :=$
 $Risorse_allocate[processo_richiedente, k] + Nuova_richiesta[k];$
 $Risorse_totali_allocate[k] := Risorse_totali_allocate[k] + Nuova_richiesta[k];$

Gli *input* li abbiamo già visti.

Active è l'insieme dei processi attivi; *fattibile* ci dice se una richiesta è ammissibile; *Nuova_richiesta* tiene conto delle nuove richieste fatte da un processo; *Allocazione_simulata* e *Risorse_totali_allocate_simulate* le usiamo per la simulazione.

Possiamo dividere tale algoritmo in **5 passi**:

1. Individuiamo tutti i processi attivi nel sistema, che altri non sono che tutti i processi che sono in stato *running* o in stato *blocked*.
Raccoglierò poi le nuove richieste che ciascun processo può fare per ogni classe di risorse. gli indici di *Risorse_richieste* saranno l'ID del processo richiedente e l'ID di ogni classe delle risorse che lui richiede.
2. Verifichiamo se la richiesta arrivata è una richiesta fattibile. Non lavoriamo però direttamente sulle risorse allocate ma uso l'array *Allocazione_simulata*, che all'inizio di tale passo è una copia di *Risorse_allocate*.
Per ogni classe di risorse del processo che ha fatto la richiesta verificherò se tale allocazione è fattibile andando ad aggiornare l'allocazione delle risorse, dove l'aggiornamento sarà dato dall'allocazione corrente + la nuova richiesta.
Ovviamente, questo dovrà farlo sia per *Allocazione_simulata* che *Risorse_totali_allocati_simulate*.
3. Verifichiamo se la *simulazione* sia fattibile. Supponiamo che sia così, poniamo dunque *fattibile* = *true*.
Per verificare se lo stato di allocazione *simulato* è fattibile, dobbiamo verificare *Risorse_totali >= Risorse_totali_allocati_simulate* per ogni classe.
Se così non fosse, ovvero se c'è anche una sola classe di risorse per cui *Risorse_totali < Risorse_totali_allocati_simulate* allora la simulazione non è fattibile in quanto andrei ad allocare più risorse di quante ne ho, e dunque *fattibile* = *false*.
4. **(Passo fond)** Se la simulazione è fattibile, verifichiamo se è possibile costruire una sequenza **completamento-rilascio-allocazione** per ogni processo all'interno del sistema, cioè verifichiamo se “lo stato di allocazione prioiettato è uno stato di allocazione sicuro”.
Se *Active* contiene un processo *P_i* tale che, per ogni processo, si ha che:

$$Risorse_{totali_k} - Risorse_{totali_allocate_k} \geq Risorse_{massime_{l,k}} - Risorse_{allocate_{l,k}}$$

allora si tratta di un processo che ha a disposizione tutte le risorse per andare a completamento. Quindi elimino tale processo da *Active* e aggiorno *Risorse_totali_allocati_simulate[k]* levandogli *Risorse_allocate[l,k]*.

5. Verifichiamo se l'insieme *Active* sia vuoto. Se così fosse vuol dire che “lo stato di allocazione prioiettato è uno stato di allocazione sicuro” e quindi posso finalmente concedere PER DAVVERO le risorse del processo ed aggiorno tutte le strutture dati in gioco.
In caso contrario, non possiamo assicurare che tale stato sia sicuro e quindi potrebbero formarsi dei deadlock. In tal caso non eseguiamo tale passo e la richiesta del processo in questione rimane *pendente*.

Esempio 8.11 - Algoritmo del banchiere per classi di risorsa multiple

La Figura 8.8 illustra il funzionamento dell'algoritmo del banchiere in un sistema che contiene quattro processi P_1, P_2, P_3 e P_4 . Quattro classi di risorsa che contengono 6, 4, 8 e 5 unità di risorsa, di cui 5, 3, 5 e 4 unità di risorsa sono attualmente allocate. Il processo P_2 ha effettuato una richiesta $(0, 1, 1, 0)$, che sta per essere elaborata.

L'algoritmo simula l'accettazione di questa richiesta nel Passo 2, e controlla se lo stato di allocazione proiettato è sicuro nel Passo 4. La Figura 8.8(b) mostra le strutture dati dell'algoritmo del banchiere all'inizio di questo controllo. In questo stato, sono disponibili 1, 0, 2 e 1 unità di risorsa, quindi solo il processo P_1 può terminare. Di conseguenza, l'algoritmo simula il suo completamento. La Figura 8.8(c) mostra le strutture dati dopo che P_1 ha terminato. Le risorse allocate per P_1 sono state rilasciate e quindi vengono eliminate da *Risorse_allocate_simulate* e P_1 è cancellato dall'insieme *Active*. Il processo P_4 ha bisogno di 0, 1, 3 e 4 unità di risorsa per soddisfare la sua richiesta massima di risorse necessarie, quindi, ora può avere queste risorse allocate, e può terminare. I processi rimanenti possono terminare nell'ordine P_2 , P_3 . Di conseguenza, la richiesta effettuata dal processo P_2 viene accettata.

(a) Stato dopo il passo 1

	R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4
P_1	2	1	2	1	P_1	1	1	1	1	P_1	0	0	0	0
P_2	2	4	3	2	P_2	2	0	1	0	P_2	0	1	1	0
P_3	5	4	2	2	P_3	2	0	2	2	P_3	0	0	0	0
P_4	0	3	4	1	P_4	0	2	1	1	P_4	0	0	0	0
Risorse massime				Risorse allocate				Risorse richieste				Attivi		
												$\{P_1, P_2, P_3, P_4\}$		
												Totali allocate		
												$[5 \ 3 \ 5 \ 4]$		
												Totali esistenti		
												$[6 \ 4 \ 8 \ 5]$		

(b) Stato dopo il ciclo while del passo 4

P_1	$\begin{array}{ c c c c } \hline 2 & 1 & 2 & 1 \\ \hline \end{array}$	P_1	$\begin{array}{ c c c c } \hline 1 & 1 & 1 & 1 \\ \hline \end{array}$	P_1	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$	Totali allocate	$\begin{array}{ c c c c } \hline 5 & 4 & 6 & 4 \\ \hline \end{array}$
P_2	$\begin{array}{ c c c c } \hline 2 & 4 & 3 & 2 \\ \hline \end{array}$	P_2	$\begin{array}{ c c c c } \hline 2 & 1 & 2 & 0 \\ \hline \end{array}$	P_2	$\begin{array}{ c c c c } \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$	simulate	$\begin{array}{ c c c c } \hline 5 & 4 & 6 & 4 \\ \hline \end{array}$
P_3	$\begin{array}{ c c c c } \hline 5 & 4 & 2 & 2 \\ \hline \end{array}$	P_3	$\begin{array}{ c c c c } \hline 2 & 0 & 2 & 2 \\ \hline \end{array}$	P_3	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$	Totali esistenti	$\begin{array}{ c c c c } \hline 6 & 4 & 8 & 5 \\ \hline \end{array}$
P_4	$\begin{array}{ c c c c } \hline 0 & 3 & 4 & 1 \\ \hline \end{array}$	P_4	$\begin{array}{ c c c c } \hline 0 & 2 & 1 & 1 \\ \hline \end{array}$	P_4	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$	Ativi	$\{P_1, P_2, P_3, P_4\}$
Risorse massime		Risorse allocate		Risorse richieste			

(c) Stato dopo aver simulato il completamento del processo P_1

P_1	$\begin{array}{ c c c c } \hline 2 & 1 & 2 & 1 \\ \hline \end{array}$	P_1	$\begin{array}{ c c c c } \hline 1 & 1 & 1 & 1 \\ \hline \end{array}$	P_1	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$	Totali allocate simulate	$\begin{array}{ c c c c } \hline 4 & 3 & 5 & 3 \\ \hline \end{array}$
P_2	$\begin{array}{ c c c c } \hline 2 & 4 & 3 & 2 \\ \hline \end{array}$	P_2	$\begin{array}{ c c c c } \hline 2 & 1 & 2 & 0 \\ \hline \end{array}$	P_2	$\begin{array}{ c c c c } \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$	Totali esistenti	$\begin{array}{ c c c c } \hline 6 & 4 & 8 & 5 \\ \hline \end{array}$
P_3	$\begin{array}{ c c c c } \hline 5 & 4 & 2 & 2 \\ \hline \end{array}$	P_3	$\begin{array}{ c c c c } \hline 2 & 0 & 2 & 2 \\ \hline \end{array}$	P_3	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$		
P_4	$\begin{array}{ c c c c } \hline 0 & 3 & 4 & 1 \\ \hline \end{array}$	P_4	$\begin{array}{ c c c c } \hline 0 & 2 & 1 & 1 \\ \hline \end{array}$	P_4	$\begin{array}{ c c c c } \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$		
Risorse massime		Risorse allocate		Risorse richieste		Attivi	$\{P_2, P_3, P_4\}$

(d) Stato dopo aver simulato il completamento del processo P_4

P_1	2 1 2 1	P_1	1 1 1 1	P_1	0 0 0 0	Totali allocate simulate	4 1 4 2
P_2	2 4 3 2	P_2	2 1 2 0	P_2	0 1 1 0		
P_3	5 4 2 2	P_3	2 0 2 2	P_3	0 0 0 0	Totali esistenti	6 4 8 5
P_4	0 3 4 1	P_4	0 2 1 1	P_4	0 0 0 0		
Risorse massime		Risorse allocate		Risorse richieste		Attivi	$\{P_2, P_3\}$

(e) Stato dopo aver simulato il completamento del processo P_2

P_1	2 1 2 1	P_1	1 1 1 1	P_1	0 0 0 0	Totali allocate simulate	2 0 2 2
P_2	2 4 3 2	P_2	2 1 2 0	P_2	0 1 1 0	Totali esistenti	6 4 8 5
P_3	5 4 2 2	P_3	2 0 2 2	P_3	0 0 0 0		
P_4	0 3 4 1	P_4	0 2 1 1	P_4	0 0 0 0		
Risorse massime		Risorse allocate		Risorse richieste		Attivi	$\{P_3\}$

Algoritmo del Banchiere con allocazioni parametriche

Tale algoritmo può essere usato per risolvere alcuni problemi che hanno a che fare con le allocazioni parametriche. Quando viene posto un problema di questo tipo di solito viene chiesto di **determinare** l'intervallo delle variabili in cui il sistema si trova in uno stato sicuro (se esiste), o se la richiesta per un determinato processo può essere soddisfatta.

Vediamo un **esempio**:

Si supponga che in un sistema multiprocessore a 3 processori siano presenti cinque processi P_0, P_1 (allocati sul processore 0), P_2 (allocato sul processore 1) e P_3, P_4 (allocati sul processore 2) e un insieme di risorse di quattro tipi diversi A, B, C, D, e di trovarsi nella seguente configurazione:

	A	B	C	D
P_0	4	$X - 1$	3	2
P_1	8	0	$Y - 2$	2
P_2	4	0	0	0
P_3	0	0	3	2
P_4	2	1	$Z + 1$	4

Risorse allocate

	A	B	C	D
P_0	6	4	5	6
P_1	10	7	6	8
P_2	6	2	0	8
P_3	0	3	4	2
P_4	9	1	6	9

Risorse massime

	A	B	C	D
	2	2	10	4

Risorse disponibili

Determinare gli intervalli dei valori interi di X, Y e Z per i quali:

- il sistema si trova in uno stato sicuro, elencando eventuali sequenze sicure;
- la richiesta attuale di P_2 (2, 0, 0, 2) può essere soddisfatta.

Soluzione

Bisogna determinare una sequenza sicura in funzione dei parametri X, Y e Z:

- l'unico processo che può essere soddisfatto è P_0 perché:

$$\begin{cases} 5 - x \geq 0 \\ 5 - x \leq 2 \end{cases} \Rightarrow \begin{cases} x \leq 5 \\ x \geq 3 \end{cases} \Rightarrow 3 \leq x \leq 5$$

quindi:

$$P_0[6, x + 1, 13, 6]$$

- solo la richiesta di P_3 può essere soddisfatta:

$$P_3[6, x + 1, 16, 8]$$

- si può soddisfare la richiesta di P_2 :

$$P_2[10, x + 1, 16, 8]$$

- si può soddisfare la richiesta di P_4 se:

$$\begin{cases} 5 - z \geq 0 \\ 5 - z \leq 16 \end{cases} \Rightarrow \begin{cases} z \leq 5 \\ z \geq 5 - 16 = -11 \end{cases} \Rightarrow -11 \leq z \leq 5$$

quindi:

$$P_4[12, x + 2, 17 + z, 12]$$

- resta solo P_1 la cui richiesta può essere soddisfatta solo se:

$$\begin{cases} 8 - y \geq 0 \\ 8 - y \leq 17 + z \end{cases} \Rightarrow \begin{cases} y \leq 8 \\ y \geq -5 - z \end{cases} \Rightarrow -14 \leq y \leq 8$$

La sequenza sicura è dunque:

$$< P_0, P_3, P_2, P_4, P_1 > \text{ con } 3 \leq x \leq 5, -11 \leq z \leq 5, -14 \leq y \leq 8$$

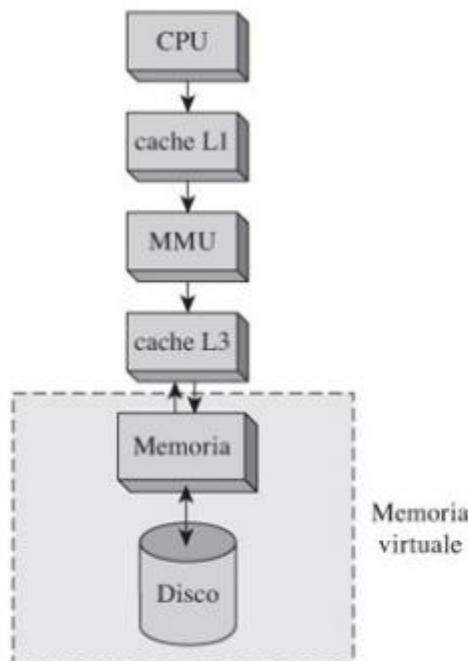
La richiesta di $P_2[2, 0, 0, 2]$ non può essere immediatamente soddisfatta in quanto porterebbe il sistema in uno stato non sicuro. Può essere soddisfatta dopo che P_0 e P_3 hanno rilasciato le loro risorse. Nota: il fatto che il sistema sia multiprocessore non influenza l'esecuzione dei processi in quanto non ci sono processi che possono ottenere le risorse contemporaneamente.

LEZIONE 17 – GESTIONE DELLA MEMORIA (1)

GERARCHIA DELLA MEMORIA

La memoria è organizzata come una gerarchia. La CPU fa riferimento alla memoria più veloce quando deve accedere a un'istruzione o a un dato, che è la *cache* e si trova sul processore stesso. Se l'istruzione o il dato richiesto non è disponibile in cache, viene prelevato dal livello successivo della gerarchia di memoria, che potrebbe essere una *cache più lenta* o la *memoria* (RAM). Se l'istruzione o il dato richiesto non è disponibile nemmeno al livello successivo della gerarchia di memoria, viene prelevato da un livello ancora inferiore, come il *disco*.

Le *prestazioni* di un processo dipendono dagli hit ratio ai vari livelli della gerarchia di memoria, dove l'**hit ratio** indica la frazione di istruzioni o dati effettivamente presenti.



Livelli	Gestione	Prestazioni
Cache	Allocazione e uso gestiti in hardware	Garantire hit ratio elevati
Memoria	Allocazione gestita dal kernel ed utilizzo della memoria allocata gestita dalla libreria run-time	(1) Mantenere più processi in memoria, (2) Garantire hit ratio elevati
Disco	Allocazione ed uso gestiti dal kernel	Caricamento e memorizzazione veloci di parti dello spazio di indirizzamento di un processo

Binding degli indirizzi

I programmi su disco, pronti per essere portati in memoria, costituiscono una coda di Input. Senza alcun supporto dovrebbero essere caricati all'indirizzo 0000, tuttavia sarebbe ideale che possano essere eseguiti in *qualsiasi* parte della memoria.

Quello che avviene è che, durante la costruzione dell'eseguibile, gli **indirizzi** sono rappresentati in modi diversi:

- Gli *indirizzi del codice sorgente* sono solitamente simbolici, e viene usata ad esempio una variabile *count*;
- Gli *indirizzi del codice compilato* devono essere associati (**bind**) ad *indirizzi rilocabili*, cioè il loro indirizzo deve essere calcolato da una certa posizione di un certo modulo.

Il linker o il loader associano (bind) gli indirizzi rilocabili ad *indirizzi assoluti*.

Ogni associazione ci fa passare da uno spazio di indirizzi ad un altro.

Associazione (binding) di Istruzioni e dati in Memoria

L'associazione degli indirizzi delle istruzioni e dei dati agli indirizzi di memoria può avvenire in tre fasi diverse:

- **Compilazione**: se la posizione di memoria è nota a priori, può essere generato un codice assoluto; se però la posizione di partenza cambia, deve essere ricompilato;
- **Caricamento**: genera codice rilocabile se la posizione di memoria non è nota in fase di compilazione;
- **Esecuzione**: il processo può essere spostato durante l'esecuzione da un segmento di memoria all'altro; tuttavia quest'opzione necessita di supporto hardware per il mapping degli indirizzi.

ALLOCAZIONE STATICÀ E DINAMICA DELLA MEMORIA

L'allocazione della memoria è un particolare aspetto del *binding*.

Il **binding** può essere di due tipi:

- **statico**: un binding eseguito prima dell'esecuzione di un programma;
- **dinamico**: un binding eseguito durante l'esecuzione del programma.

L'*allocazione statica* è eseguita dal compilatore, linker o loader. Ciò comporta che la dimensione delle strutture dati devono essere note a priori.

L'*allocazione dinamica* fornisce più flessibilità, in quanto le azioni nell'allocazione di memoria costituiscono un overhead durante le operazioni.

Spazi degli indirizzi Logico e Fisico, e MMU

L'indirizzamento è diviso in una parte *logica* ed una *fisica*:

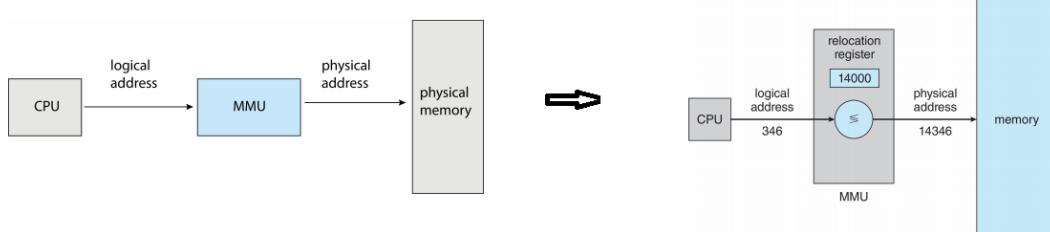
- **Indirizzo logico** (virtuale): Un *indirizzo logico* è l'indirizzo di un'istruzione o dato usato in un processo.
È generato dalla CPU. Si tratta dell'insieme di tutti gli indirizzi logici generati da un programma (*variabili, vettori, ecc...*), che costituiscono il suo *spazio di indirizzamento logico*.
- **Indirizzo fisico**: Un *indirizzo fisico* è l'indirizzo di memoria dove è memorizzata un'istruzione o un dato.
È l'indirizzo visto dall'unità di memoria. Si tratta dell'insieme di tutti gli indirizzi fisici generati da un programma, che costituiscono il suo *spazio di indirizzamento fisico*.

L'**MMU** è un componente HW che associa lo spazio di indirizzamento logico a quello fisico a run-time (lo converte).

Grazie a questa separazione è possibile ricaricare il processo in una qualsiasi zona di memoria libera in quel momento.

L'MMU effettua tale conversione utilizzando i *registri base*, in questo contesto chiamati *registri di rilocazione*. Il **valore** nel registro di rilocazione è **aggiunto** ad ogni indirizzo generato da un processo utente nel momento in cui è inviato a memoria.

ES.



ESECUZIONE DEI PROGRAMMI

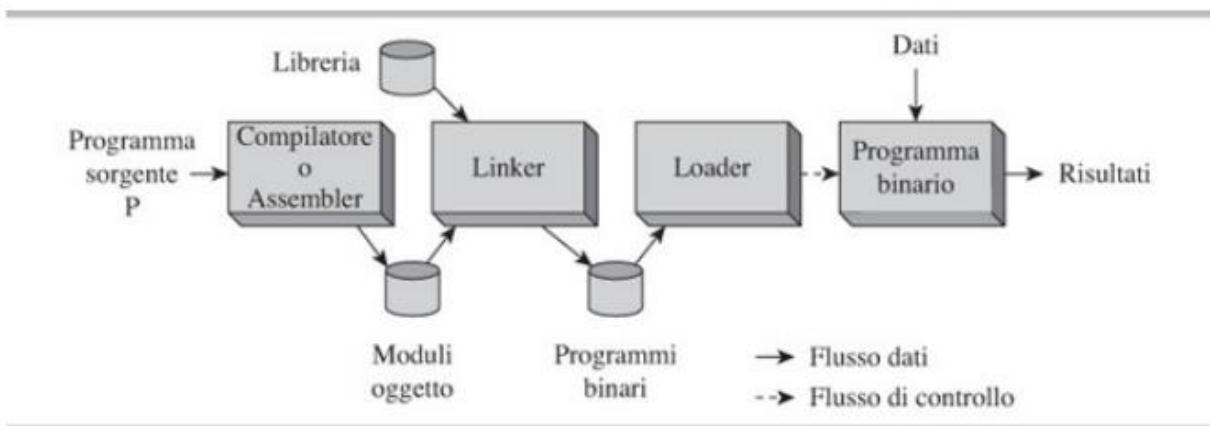
Un programma P deve essere trasformato prima di essere eseguito. Molte di queste trasformazioni effettuano il memory binding; ognuna effettua il collegamento delle istruzioni e i dati del programma a un nuovo insieme di indirizzi.

Possiamo distinguere **3 trasformazioni**:

- *Compilazione o assemblaggio*: un compilatore o un assembler è chiamato genericamente traduttore. Traduce il programma P in un programma equivalente nella forma di modulo oggetto. Questo programma contiene le istruzioni in linguaggio macchina del computer. Nell'invocare il compilatore, l'utente specifica l'origine del programma (non avviene quasi mai) altrimenti, il compilatore assume un indirizzo di default, tipicamente 0.
- *Linker*: include gli indirizzi delle librerie all'interno del codice oggetto.
- *Loader*: carica l'eseguibile in una specifica allocazione di memoria, in base al fatto che il *linking* sia stato effettuando in modo *statico* o *dinamico*, e di conseguenza anche il *loading* può avvenire in modo *statico* o *dinamico*.
Nel caso avvenga in modo *dinamico* è necessario effettuare una *rilocazione*.

Gli indirizzi assegnati dal compilatore sono chiamati *indirizzi tradotti*.

Gli indirizzi assegnati dal linker sono chiamati *indirizzi linkati*, il quale si occupa delle funzioni di rilocazione e del linking.



Rilocazione

Per eseguire un programma, il kernel gli assegna un'area grande abbastanza da contenerlo e richiama il LOADER con il nome del programma e l'origine di caricamento come parametri.

Distinguiamo **due tipi** di rilocazione:

- **statica**: la rilocazione viene eseguita prima dell'esecuzione del programma. Cioè, gli indirizzi effettivi vengono calcolati solo al momento del caricamento del programma in memoria centrale per l'esecuzione e quindi possono variare da un'esecuzione all'altra. Una volta stabiliti restano tali.
- **dinamica**: la rilocazione viene eseguita durante l'esecuzione del programma. Cioè, gli indirizzi possono essere tradotti durante l'esecuzione del processo, per riconfigurare la memoria in caso di evenienze sopravvenienti.
Si tratta della modalità più complessa e sofisticata.

La rilocazione **dinamica** può avvenire in due modi:

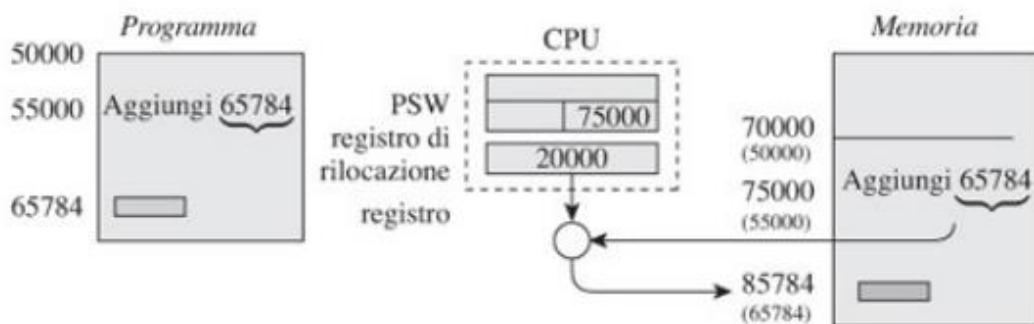
- si **sospende** l'esecuzione del processo, lo si **riallocava** come se fosse una riallocazione statica, e ne si **riprende** l'esecuzione.
Tuttavia durante l'esecuzione il sistema dovrebbe essere a conoscenza degli indirizzi delle varie fasi (*compiling, linking e loading*) e si introduce overhead per la sospensione e ripresa del processo. Per tal motivo tale strategia non è adottata.
- si utilizzano i **registri di rilocazioni**.

esempio rilocazione dinamica con registri di rilocazione:

Supponiamo che stiamo eseguendo un programma allocato dalla locazione 50.000, e supponiamo di star eseguendo una certa istruzione alla locazione 55.000 che fa riferimento all'indirizzo 65.784.

Supponiamo che, per un qualsiasi motivo, il programma venga rilocato e che adesso parta da 70.000.

Quello che avviene è che il *registro di rilocazione* conterrà il valore 20.000, questo perché è il valore che deve essere sommato al vecchio indirizzo del programma per ottenere quello nuovo. (di conseguenza, l'istruzione sarà locata a 75.000 e l'istruzione a cui farà riferimento sarà ora 85.784).



Loading Dinamico

Non è necessario che l'intero programma risieda in memoria per essere eseguito, ma anzi quello che si fa per rendere più efficiente la gestione della memoria è utilizzare un **loading dinamico**.

Una routine non è caricata fin quando non è invocata, in tal modo ho un miglior utilizzo dello spazio di memoria. Tutte le routine sono tenute su disco in formato di caricamento rilocabile.

È utile quando sono necessarie grandi quantità di codice per gestire casi che si verificano di rado.

Inoltre, non è richiesto un supporto speciale del SO, in quanto è implementato mediante la progettazione del programma.

Linking

Distinguiamo due tipi di linking:

- **statico**: le librerie di sistema e il codice del programma sono combinati dal loader nell'immagine del programma binario;
- **dinamico**: il linking è posticipato fino all'esecuzione.

Nel caso del *linking dinamico* vengono utilizzati dei piccoli pezzi di codice, chiamati **stub**, per localizzare le opportune routine di libreria residenti in memoria.

Il linking dinamico è noto anche come **librerie condivise**, in quanto se più processi usano la stessa libreria, invece di utilizzare il linking statico con cui creeremmo una copia di quella libreria per ogni processo, usiamo il linking dinamico così da creare un'unica copia di quella libreria che possa essere usata dai processi.

ALLOCAZIONE DI MEMORIA AD UN PROCESSO: STACK ed HEAP

Il compilatore di un linguaggio di programmazione genera il codice di un programma, alloca i suoi dati statici e crea un modulo oggetto del programma.

Il linker linka il programma con le funzioni di libreria e il supporto run-time del linguaggio di programmazione, prepara un eseguibile del programma e lo memorizza in un file.

L'informazione relativa alla dimensione del programma è memorizzata nell'elemento della directory relativo all'eseguibile.

Il supporto run-time alloca **due tipi di dati** durante l'esecuzione del programma.

- Il primo tipo di dati riguarda tutti quei dati che sono allocati quando si opera all'interno di una funzione, procedura o blocco e sono deallocati al termine della relativa esecuzione. A causa del modo di allocazione/deallocazione LIFO, i dati sono allocati sullo **stack**.
- Il secondo tipo di dati viene creato dinamicamente da un programma utilizzando particolari funzioni, come la funzione *new* del C++. Ci riferiamo a tali dati come *dati dinamici controllati* dal programma (PCD), i quali sono allocati utilizzando un **heap**.

Stack

In uno stack, le allocazioni e le deallocazioni sono eseguite secondo la modalità last-in first-out (LIFO) in corrispondenza, rispettivamente, alle operazioni *push* e *pop*.

In tale struttura dati l'allocazione di memoria è **contigua**.

Un puntatore chiamato *stack base* (SB) punta al primo elemento dello stack, mentre un puntatore chiamato *top of stack* (TSO) punta all'ultimo elemento.

L'insieme di elementi dello stack che fanno riferimento a una chiamata di funzione è chiamato *stack frame*, il quale contiene gli indirizzi o i valori dei parametri della funzione e l'indirizzo di ritorno.

Durante l'esecuzione della funzione vengono creati i dati locali della funzione all'interno dello stack frame stesso. Al termine dell'esecuzione, l'intero stack frame viene estratto dallo stack e l'indirizzo di ritorno contenuto al suo interno è utilizzato per passare il controllo al programma chiamante.

Inoltre, per facilitare l'uso di uno stack frame vengono utilizzati altri 2 elementi:

- il *Frame Base* (FB) che punta all'inizio dello stack frame;
- un puntatore al *precedente stack frame*.

Heap

Un heap consente di allocare e deallocare la memoria in ordine **casuale**. Una richiesta di allocazione da parte di un processo ritorna un puntatore all'area di memoria allocata nell'heap e il processo accede all'area di memoria allocata attraverso questo puntatore. Una richiesta di deallocazione deve fornire un puntatore all'area di memoria da deallocare.

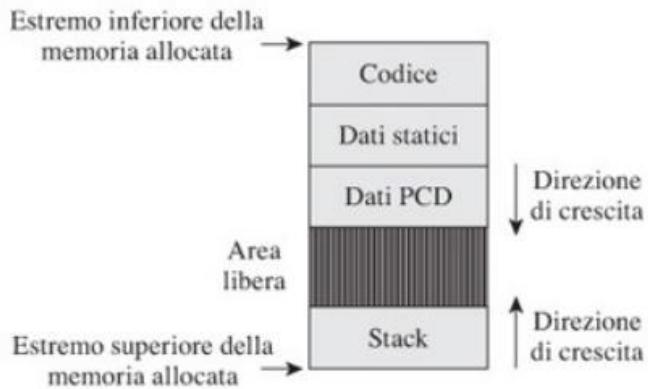
Viene utilizzato per gestire i *dati dinamici controllati* dal programma (PCD).

Modello di allocazione della memoria

Alla creazione di un nuovo processo, il kernel deve decidere quanta memoria allocare alle seguenti componenti:

- codice e dati statici del programma;
- stack;
- dati dinamici controllati da programma (dati PCD).

Il kernel non sa quanto spazio allocare per i dati PCD e lo stack, per tal motivo heap e stack condividono la stessa area di memoria dove però crescono in direzioni opposte.



Protezione della memoria

La protezione della memoria usa i registri **base** e **size**, in modo tale che per ciascun processo conosciamo l'indirizzo logico di partenza e la sua dimensione.

Verificheremo che l'indirizzo usato non si trova al di fuori del *range di memoria* e che non ci siano interferenze fra processi.

GESTIONE DELLO HEAP

La gestione dell'heap deve **ottimizzare** la gestione della memoria, ovvero si deve ottimizzare lo spazio cercando contemporaneamente di ottimizzare la velocità di allocazione.

In un heap, il riuso della memoria non è automatico, ma possono essere fatte due cose:
Riuso della memoria e Buddy system.

1. RIUSO DELLA MEMORIA

Per il riuso della memoria possiamo distinguere **3 funzioni del kernel:**

Funzione	Descrizione
Mantenere una free list	La <i>free list</i> contiene informazioni relative a ogni area di memoria libera. Quando un processo libera parti di memoria, le informazioni relative alla memoria liberata vengono inserite nella free list. Quando un processo termina, ogni area di memoria a esso allocata e le informazioni relative vengono inserite nella free list.
Selezionare un'area di memoria per l'allocazione	Quando viene effettuata una nuova richiesta di memoria, il kernel seleziona l'area di memoria più adatta per soddisfare la richiesta.
Unire le aree di memoria libere	Due o più aree di memoria libere contigue possono essere unite per formare una sola grande area libera. Le aree da unire sono rimosse dalla free list e viene inserita al loro posto l'area più grande creata.

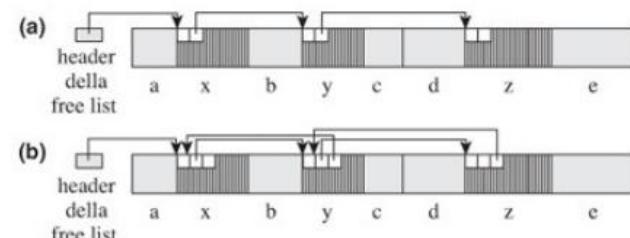
Gestione delle Free List

Per ogni area di memoria nella *free list*, il kernel gestisce:

- dimensione dell'area di memoria;
- puntatori usati per formare la lista.

Tale lista viene costruita in modo tale che all'interno di ciascuna area libera viene riservato dello spazio per rappresentare le dimensioni e un puntatore alla successiva area libera della lista.

La Free List può essere gestita con *singolo puntatore* o con *doppio puntatore*.



Gestione dello spazio libero: (a) free list con singolo puntatore;
(b) free list con doppio puntatore.

Tenendo in considerazione una richiesta per n byte di memoria, posso usare delle strategie che si basano sul fatto di suddividere l'area di memoria in due parti: n byte saranno allocati per la richiesta, mentre la restante parte verrà reinserita nella *free-list*.

Possiamo distinguere **tre strategie di allocazione**:

- **First-fit:** usa la prima area sufficientemente grande; alla lunga tale aree di memoria diventano sempre più piccole, fino a poter non essere più sufficienti per soddisfare future richieste (*frammentazione della memoria*).
- **Best-fit:** usa l'area sufficientemente grande più piccola; non vengono più divise aree molto grandi, tuttavia anche in questo caso alla lunga si producono molte aree piccole, ancora più piccole di quelle della First-fit. Inoltre l'operazione di ricerca di un'area di dimensioni specifiche introduce overhead.
- **Next-fit:** usa la successiva area sufficientemente grande; le aree di memoria che costituiscono la lista sono suddivise in modo un po' più uniforme.

Frammentazione della memoria

La frammentazione della memoria è un problema che riguarda l'esistenza di aree di memoria inutilizzabili in un computer.

Possiamo distinguerne di due tipi:

Tipologia di frammentazione	Descrizione
Frammentazione esterna	Alcune aree di memoria sono troppo piccole per essere allocate.
Frammentazione interna	Viene allocata più memoria rispetto a quella richiesta, dunque una parte della memoria allocata resta inutilizzata.

Vengono usate diverse tecniche per **minimizzare** la frammentazione della memoria.

La **frammentazione esterna** può essere gestita **unendo aree di memoria libera**, e si può far ciò usando due tecniche generali:

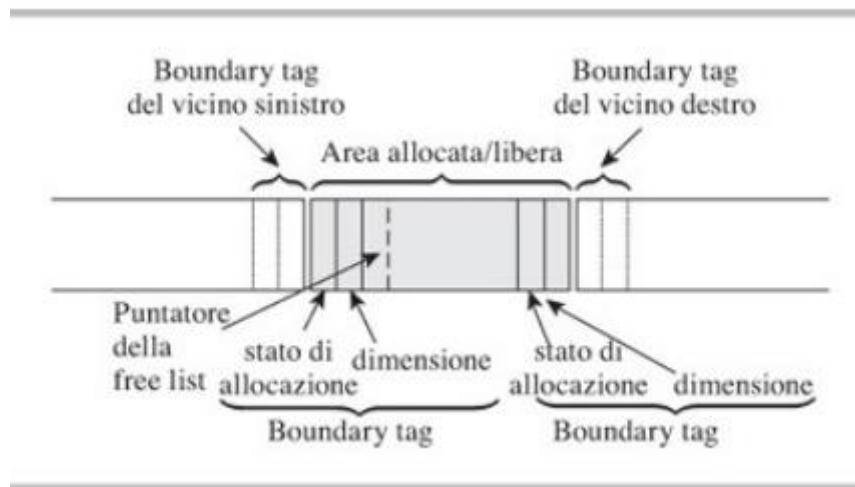
- Boundary tag;
- Compattazione della memoria.

Boundary tag

Un **tag** è un descrittore di stato di un'area di memoria. Consiste di una coppia ordinata che rappresenta lo stato di allocazione dell'area e la sua dimensione.

I **boundary tag** sono tag identici memorizzati all'inizio e alla fine dell'area di memoria, ovvero nei primi e negli ultimi byte dell'area.

Quando un'area di memoria diventa libera, il kernel controlla i boundary tag delle aree vicine. Se qualcuno dei vicini è libero, viene unito con l'area appena liberata.



Quando si usa questo metodo di unione, viene applicata una relazione chiamata **regola del 50 per cento**. Quando un'area di memoria viene rilasciata, il numero totale di aree libere nel sistema:

- aumenta di 1 se l'area liberata non ha alcun vicino;
- diminuisce di 1 se l'area liberata ha due vicini;
- non cambia se l'area liberata ha un solo vicino libero.

La regola del 50 per cento aiuta a stimare la dimensione della free list, e questo è dovuto anche al motivo per cui si chiama in questo modo:

Supponiamo di avere 3 tipi di aree occupate: A ha due aree libere vicine, B ha un'area libera vicina, C non ha alcuna area libera vicina.



$$\text{numero di aree allocate, } n = \#A + \#B + \#C$$

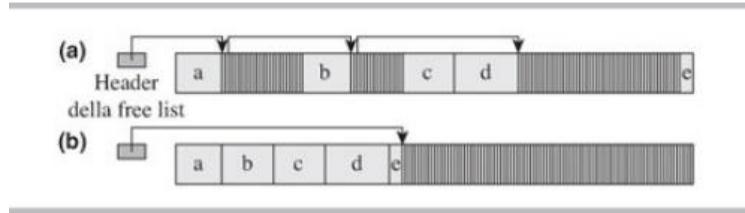
$$\text{numero di aree libere, } m = \frac{1}{2} (2 \times \#A + \#B)$$

Ovvero, il numero di aree libere è esattamente uguale al numero di aree allocate.

Compattazione

La compattazione della memoria è ottenuta impachettando tutte le aree allocate verso un'estremità della memoria.

Tale strategia è possibile solo se è fornito un **registro di rilocazione**.



2. BUDDY SYSTEM

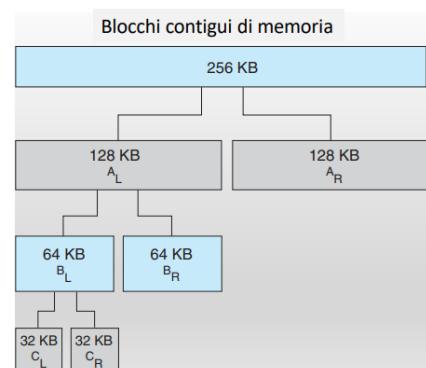
Il buddy system e gli allocatori potenza del 2 eseguono l'allocazione di memoria in blocchi di poche dimensioni predefinite.

Questa caratteristica porta alla frammentazione interna poiché parte della memoria in ogni blocco allocato può essere sprecata. Tuttavia, consente all'allocatore di mantenere free list separate per blocchi di dimensioni differenti. Questa organizzazione evita ricerche costose nella free list e porta ad allocazioni e deallocazioni veloci.

Allocazione Buddy

I blocchi creati dividendo un blocco sono chiamati blocchi **buddy**. I blocchi buddy liberi vengono uniti per ricreare il blocco da cui erano stati creati. Questa operazione si chiama fusione. Secondo questo sistema, i blocchi liberi contigui che non sono buddy non sono fusi.

Un esempio di buddy system è il *buddy system binario*:



Allocatori potenze di 2

La dimensione dei blocchi di memoria sono potenze di 2 e vengono mantenute *free list* separate per i blocchi di dimensione diversa (su questo aspetto è simile al Buddy system binario).

Ogni blocco contiene un header, che a sua volta contiene l'indirizzo di una free list a cui dovrebbe essere aggiunto quando diventa libero.

Quando viene fatta una richiesta viene allocato l'intero blocco (nessuna divisione di blocchi). Quando il blocco è rilasciato non avviene alcuna fusione; invece il blocco è restituito alla sua free list.

LEZIONE 18 – GESTIONE DELLA MEMORIA (2)

ALLOCAZIONE CONTIGUA DELLA MEMORIA

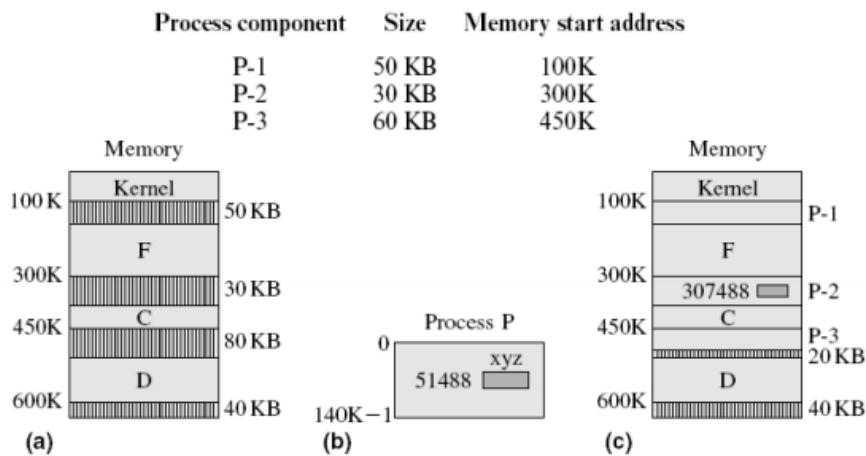
Ad ogni processo è allocata una singola area contigua di memoria.

Affronta il problema della *frammentazione della memoria* applicando tecniche di compattazione e riuso della memoria.

La compattazione richiede anche un registro di locazione. Se manca tale registro, anche lo *swapping* diviene un problema: un processo in swap-in dovrà essere allocato nello stesso blocco di memoria che gli è stato tolto in fase di swap-out, rendendo tale operazione poco flessibile e inefficiente.

ALLOCAZIONE NON CONTIGUA DELLA MEMORIA

Parti dello spazio di indirizzamento di un processo sono distribuite tra aree diverse di memoria. Tale tecnica riduce la frammentazione esterna.



Indirizzi logici e fisici, e traduzione

Il fatto che si possa allocare diverse componenti in diverse aree di memoria richiede l'uso di meccanismi speciali che consentano di far corrispondere correttamente agli indirizzi logici, gli indirizzi fisici.

Ricordando che:

- **Indirizzo logico** (virtuale): Un *indirizzo logico* è l'indirizzo di un'istruzione o dato usato in un processo.
È generato dalla CPU. Si tratta dell'insieme di tutti gli indirizzi logici generati da un programma (*variabili, vettori, ecc...*), che costituiscono il suo *spazio di indirizzamento logico*.
- **Indirizzo fisico**: Un *indirizzo fisico* è l'indirizzo di memoria dove è memorizzata un'istruzione o un dato.
È l'indirizzo visto dall'unità di memoria. Si tratta dell'insieme di tutti gli indirizzi fisici generati da un programma, che costituiscono il suo *spazio di indirizzamento fisico*.

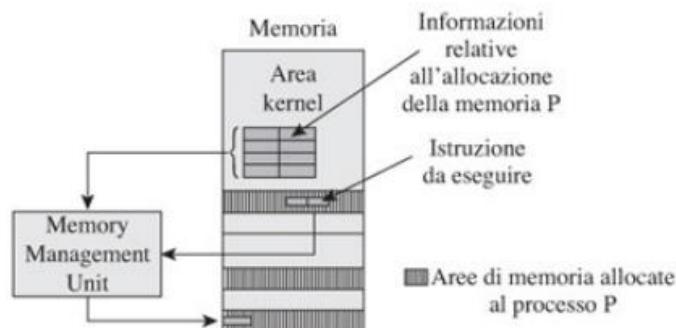
Il *kernel* memorizza le informazioni sulle aree di memoria allocate al processo P in una tabella disponibile alla MMU.

La CPU invia l'indirizzo logico di ogni dato, o istruzione, usato in P alla MMU.

La MMU usa le info sull'allocazione della tabella per calcolare l'indirizzo fisico, ovvero l'*indirizzo di memoria effettivo* del dato o dell'istruzione.

La procedura per determinare l'indirizzo di memoria effettivo è chiamata *traduzione dell'indirizzo*.

L'*indirizzo logico* di una istruzione da eseguire viene inviata, dalla CPU, alla MMU.
La MMU calcola l'indirizzo fisico



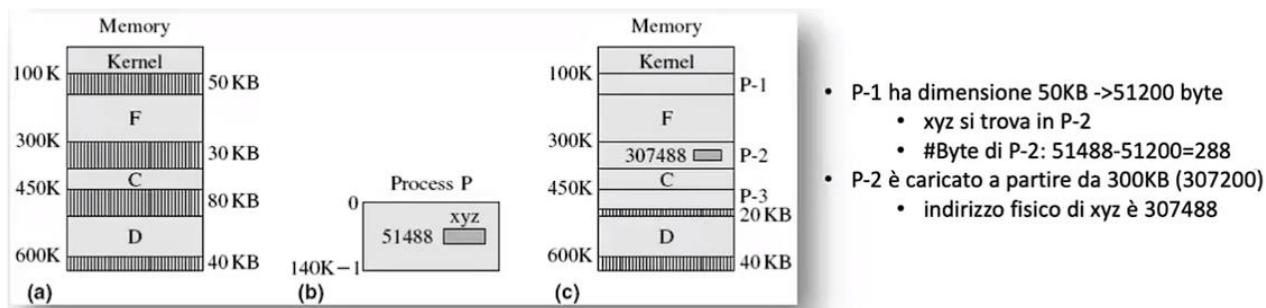
Un *indirizzo logico* si compone di due parti: l'**id della componente** del processo contenente l'indirizzo, e l'**id del byte** all'interno della componente.

Queste due parti sono viste come una **coppia (comp_i, byte_i)**.

La MMU calcola il suo indirizzo di memoria effettivo mediante la formula:

$$\begin{aligned} \text{indirizzo di memoria effettivo di } & (comp_i, byte_i) \\ &= \text{indirizzo di avvio dell'area di memoria allocata a } comp_i \\ &+ \text{numero del byte di } byte_i \text{ in } comp_i \end{aligned}$$

- Esempio: P fa riferimento all'area dati di xyz con l'indirizzo logico **(P-2, 288)**
- La MMU calcola l'indirizzo fisico effettivo come $307.200 + 288 = 307.488$



Per l'allocazione non contigua della memoria distinguiamo **due approcci**: *paginazione* e *segmentazione*.

Inoltre possiamo distinguere un ulteriore approccio ibrido: *segmentazione con paginazione*.

Prima di entrare nel dettaglio dell'allocazione non contigua, facciamo un confronto fra questa e quella contigua.

ALLOCAZIONE CONTIGUA VS NON CONTIGUA

Funzione	Allocazione contigua	Allocazione non contigua
Allocazione della memoria	Il kernel alloca una singola area di memoria a un processo.	Il kernel alloca diverse aree di memoria a un processo – ogni area contiene una componente del processo.
Traduzione dell'indirizzo	La traduzione dell'indirizzo non è necessaria.	La traduzione dell'indirizzo è eseguita dalla MMU durante l'esecuzione del programma.
Frammentazione della memoria	Si verifica frammentazione esterna quando viene usata l'allocazione first-fit, best-fit o next-fit. Si verifica frammentazione interna se l'allocazione della memoria viene eseguita in blocchi di poche dimensioni standard.	Nella paginazione non si verifica la frammentazione esterna ma può verificarsi la frammentazione interna. Nella segmentazione si verifica la frammentazione esterna, ma non si verifica la frammentazione interna.
Swapping	Se il computer non ha un registro di rilocazione, un processo swapped deve essere rimesso nell'area originariamente allocata a esso.	Le componenti di un processo swapped possono essere caricate in una qualunque parte della memoria.

PROTEZIONE DELLA MEMORIA

Le aree di memoria ad un programma devono essere protette dalle interferenze con altri programmi.

La MMU esegue questo compito con il **controllo dei limiti**: mentre esegue la traduzione per un indirizzo logico ($comp_i$, $byte_i$), controlla se $comp_i$ si trova nel programma e se $byte_i$ si trova in $comp_i$; in caso ciò non fosse vero si verifica una violazione della protezione e viene generato un interrupt.

Il controllo dei limiti può essere semplificato con la paginazione, in quanto $byte_i$ non può superare la dimensione di una pagina (quindi non può andare oltre i propri limiti, e non c'è bisogno di controllarlo).

PAGINAZIONE

Fondamentalmente, i processi consistono di componenti di dimensioni fisse chiamate *pagine*, dove la dimensione della pagina è definita dall'hardware.

Elimina la frammentazione esterna.

Da un punto di vista logico, lo **spazio degli indirizzi** di un processo è visto come un'organizzazione lineare delle **pagine**.

Ogni pagina contiene **s** byte, con **s** potenza di 2. Il valore di **s** è specificato dall'architettura del sistema.

I processi usano indirizzi logici numeri. La MMU scomponete l'indirizzo logico nella coppia (p_i, b_i) , dove p_i rappresenta il numero di pagina e b_i rappresenta l'offset in byte, e con $p_i \geq 0$ e $0 \leq b_i < s$.

La seconda condizione ci dice che lo scostamento in byte non può essere superiore ad una pagina, ma deve sempre trovarsi al suo interno.

La **memoria fisica** è divisa in aree chiamate *frame*, numerati a partire da 0.

Un **frame** ha la stessa dimensione di una **pagina**.

Il kernel mantiene una lista, chiamata *lista dei frame liberi*, per tenere traccia dei numeri di frame liberi. Quando carica un processo, il kernel consulta la lista dei frame liberi e alloca un frame libero a ogni pagina del processo.

Per facilitare la traduzione degli indirizzi, il kernel costruisce una *tavella delle pagine* (PT) per ciascun processo.

La PT ha un'entrata per ogni pagina del processo che indica il frame allocato alla pagina. Mentre esegue la traduzione per un indirizzo logico (p_i, b_i) , la MMU usa il numero di pagina p_i per:

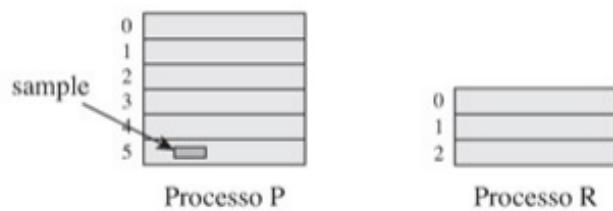
- indicizzare la PT del processo;
- ottenere il numero di frame allocato a p_i ;
- calcolare l'indirizzo di memoria effettivo.

Es.

Si considerino due processi P ed R in un sistema che utilizza una pagina di dimensione 1 KB, e dove P ha dimensione 5500 byte ed R ha dimensione 2500 byte.

Ricordando che 1KB corrispondono a 1024 byte, per contenere il processo P avremo bisogno di 6 pagine, e dove l'ultima pagina contiene solo 380 byte, e il processo R avrà bisogno di 3 pagine.

Le pagine sono numerate da 0 a n-1. Se un dato sample ha indirizzo 5248, ovvero $5 \times 1024 + 128$, la MMU vede il suo indirizzo come la coppia (5, 128), ovvero pagina 5 con offset 128.

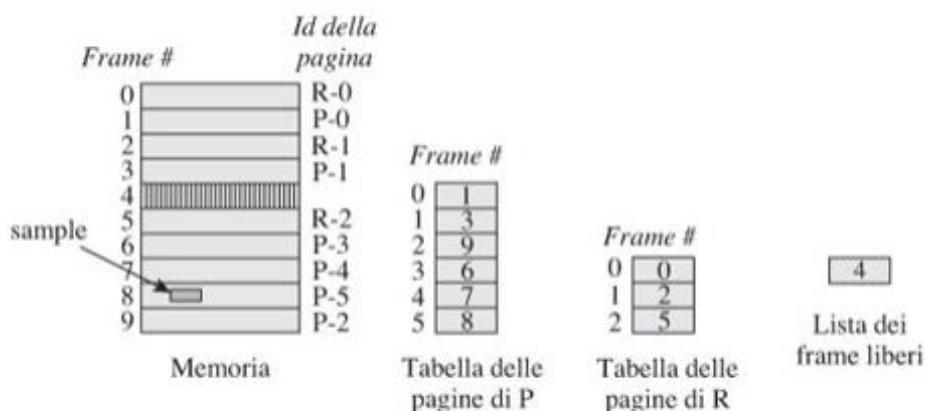


In una PT, il numero di pagina x indica in quale frame è contenuta la pagina x .

Ad esempio osserviamo la coppia (5, 128). Sappiamo che la pagina 5 è contenuta nel frame 8, dal quale determiniamo l'indirizzo di partenza del frame 8 a cui aggiunge lo scostamento.

L'indirizzo di partenza sarà semplicemente calcolato come 5×1024 .

L'indirizzo fisico di sample sarà 8320.



Traduzione (effettiva) di un indirizzo nella pagina

In realtà la MMU non fa tutti questi calcoli, ma gli basta invece fare una concatenazione di bit per ottenere sia indirizzi logici che fisici, e questo perché le pagine sono multiplo di 2.

- Notazione per la traduzione degli indirizzi

s dimensione di una pagina

l_l lunghezza di un indirizzo logico (cioè, numero di bit)

l_p lunghezza di un indirizzo fisico

n_b numero di bit usato per rappresentare il numero del byte in un indirizzo logico

n_p numero di bit usato per rappresentare il numero di pagina in un indirizzo logico

n_f numero di bit usato per rappresentare il numero di frame in un indirizzo fisico

- La dimensione di una pagina, s , è una potenza di 2

• n_b è scelto in modo che $s = 2^{n_b}$



La MMU ottiene i valori di p_i e b_i semplicemente concatenando i bit di p_i e b_i .

L'indirizzo di memoria fisico di ottiene concatenando i bit di q_i e b_i .

Questo grazie al fatto che le *pagine* e i *frame* hanno la stessa dimensione, e che queste siano potenze di 2, e che si conosce la dimensione dell'indirizzo logico n_b e dell'indirizzo fisico n_f .

Es.

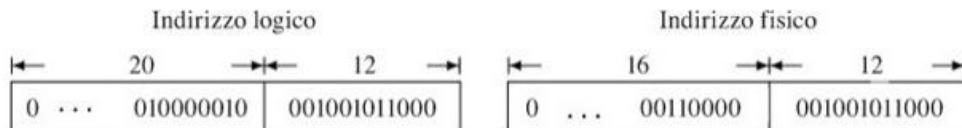
Supponiamo di avere una macchina con indirizzi logici a 32 bit ($l_l = 32$).

La dimensione di una pagina è 4KB. Sapendo che $2^{12} = 4\text{KB}$, 12 bit sono adeguati per indirizzare i byte in una pagina. Posso usare i restanti 20 bit per rappresentare il numero di pagina. ($n_p = 20$, $n_b = 12$).

La dimensione della memoria è di 256 MB, rappresentabile con 28 bit ($l_p = 28$).

Sapendo che $n_b = 12$, posso usare i restanti 16 bit (28-12) per rappresentare il numero di frame ($n_f = 16$).

Se supponiamo che alla pagina 130 sia allocato il frame 48, quindi $p_i = 130$ e $q_i = 48$, e se supponiamo che $b_i = 600$, allora gli indirizzi logici e fisici sono:



Alla MMU arriverà la coppia (130, 600), cioè l'indirizzo logico.

Per tradurlo in indirizzo fisico, dal numero di pagina si risale al numero di frame (48), e a questo gli concatena 600.

SEGMENTAZIONE

Fondamentalmente, il programmatore identifica delle entità logiche in un programma.

Esso semplifica la condivisione di codice, dati e moduli di programma tra processi.

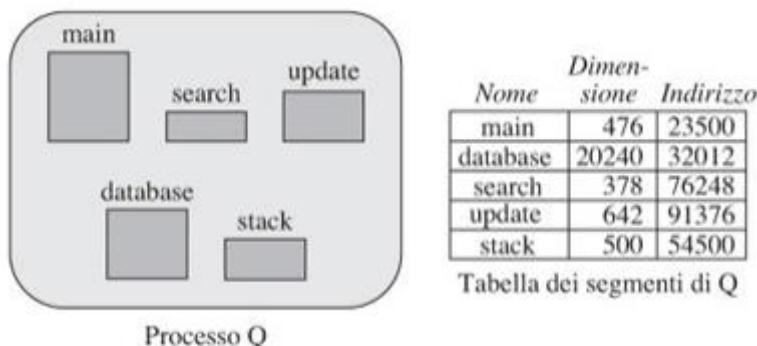
I segmenti hanno dimensioni diverse quindi il kernel usa tecniche di riuso, che può portare a frammentazione esterna.

Un *segmento* è un'entità logica in un programma, per esempio una funzione, una struttura dati o un oggetto.

Il compilatore assegna ad ogni entità un segmento di memoria che avrà una certa dimensione e un certo indirizzo di partenza.

Il kernel mantiene una *tavella di segmenti* per ogni processo Q.

Ogni indirizzo logico usato in Q è visto come una **coppia** (s_i, b_i), dove s_i è il numero del segmento e b_i è l'offset in byte del segmento.



I segmenti sono di dimensioni **variabili**, ciò significa che:

- si può avere *frammentazione esterna*;
- non si può applicare la concatenazione per determinare gli indirizzi, ma devo calcolarmi l'indirizzo di memoria effettivo attraverso delle somme.

Il kernel mantiene una **free list**. Nel caricare un processo, cerca in questa lista per eseguire l'allocazione first-fit o best-fit per ogni segmento del processo.

Es.

Se in **update** c'è l'istruzione **get_sample** al byte 232.

(update, get_sample) ha indirizzo di memoria effettivo: $91376 + 232 = 91608$

SEGMENTAZIONE CON PAGINAZIONE

La segmentazione semplifica la condivisione di codice, dati e moduli di programma tra processi, ma allo stesso tempo può portare a frammentazione esterna.

Per tal motivo si può adottare una *segmentazione con paginazione*.

Ogni segmento in un programma è **paginato separatamente**, cioè all'interno di un segmento le varie porzioni del segmento sono attribuite ad un certo **numero intero di pagine**.

Semplifica l'allocazione della memoria e la velocizza.

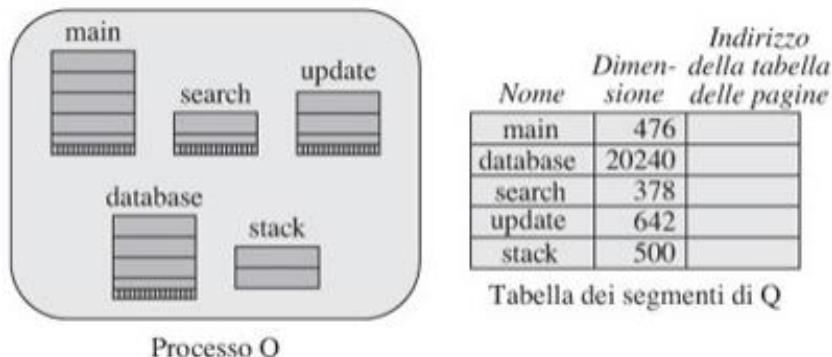
Posso avere un po' di frammentazione interna solo nell'ultima pagina che viene allocata a quello specifico segmento.

Per ogni segmento è costruita una tabella delle pagine. L'indirizzo della tabella delle pagine è mantenuto nell'entrata del segmento della tabella dei segmenti.

La **traduzione** di un indirizzo logico (s_i, b_i) è fatta in due passi:

1. Dall'entrata di s_i nella *tabella dei segmenti* è determinato l'indirizzo della tabella delle pagine.
2. Il numero del byte b_i è diviso nella coppia (ps_i, bs_i) dove ps_i è il numero di pagina del segmento s_i e bp_i è il numero del byte nella pagina p_i .

Il calcolo dell'indirizzo effettivo è fatto come nella paginazione.



confronto paginazione/segmentazione

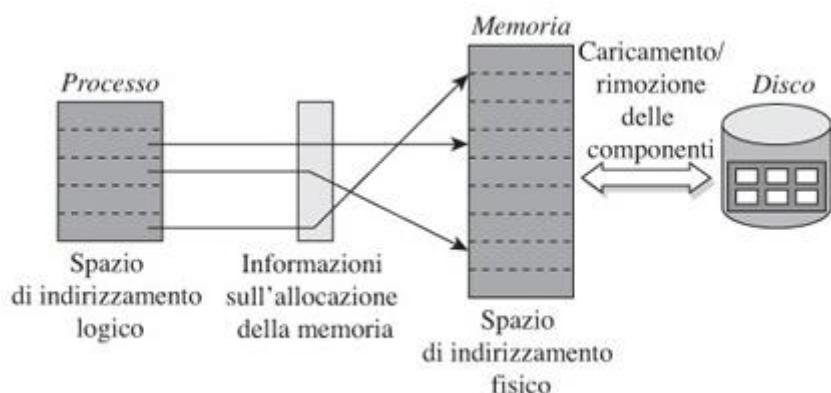
Oggetto	Confronto
Concetto	Una pagina è una porzione a misura fissa di uno spazio di indirizzamento di un processo identificato dall'hardware della memoria virtuale. Un segmento è un'entità logica all'interno di un programma, ossia una funzione, una struttura dati o un oggetto. I segmenti sono identificati dal programmatore.
Dimensione delle componenti	Tutte le pagine sono della stessa dimensione. I segmenti possono essere di dimensioni differenti.
Frammentazione esterna	Non si verifica durante la paginazione perché la memoria è divisa in page frame la cui dimensione è uguale alla dimensione delle pagine. Si verifica nella segmentazione perché un'area libera di memoria può essere piccola per ospitare un segmento.
Frammentazione interna	Si verifica nell'ultima pagina di un processo durante la paginazione. Non si verifica nella segmentazione perché un segmento è allocato in un'area di memoria la cui dimensione è uguale alla dimensione del segmento.

LEZIONE 20 – MEMORIA VIRTUALE (1)

La memoria virtuale è una gerarchia di memoria, composta dalla memoria del sistema di elaborazione e da un disco, che consente a un processo di funzionare con soltanto alcune porzioni del suo spazio di indirizzamento in memoria.

La memoria virtuale è utilizzata per due **motivi**:

- possibilità di eseguire programmi la cui dimensione più anche eccedere la memoria fisica del calcolatore;
- diminuire il carico di memoria di ciascun processo, in modo tale da averne di più in memoria e quindi eseguirne di più.



La MMU traduce gli indirizzi logici in indirizzi fisici.

Il gestore della memoria virtuale è un componente software che si occupa di gestire il *caricamento su richiesta* (si attiva quando non è presente una componente in memoria).

Tale gestore carica in memoria solo una componente dello spazio di indirizzamento di un processo per iniziare, cioè la componente che contiene l'indirizzo di avvio.

Le altre componenti sono caricate solo quando necessarie (caricamento su richiesta).

Per limitare l'impegno di memoria per un processo, il gestore rimuove componenti del processo dalla memoria di volta in volta. Le componenti saranno ricaricate quando di nuovo necessario.

Le prestazioni di un processo in memoria virtuale dipendono dal tasso a cui è necessario caricare le sue componenti in memoria.

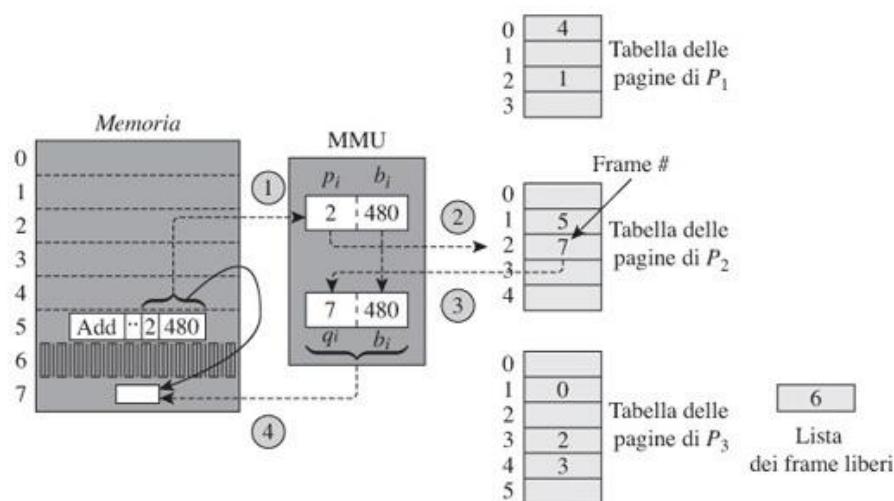
Il gestore sfrutta il principio di *località dei riferimenti* per ottenere tassi bassi.

MEMORIA VIRTUALE con PAGINAZIONE

La MMU esegue la traduzione degli indirizzi usando la tabella delle pagine.

Il procedimento svolto dalla MMU è praticamente analogo alla normale paginazione: prende il **numero di pagina** p_i e lo **scostamento** b_i e vede a quale frame appartiene il numero di pagina. Individuato il frame, lo concatena con lo scostamento ed ottiene così l'indirizzo fisico dell'operando.

In questo caso però può capitare il frame di un certo numero di pagina non sia ancora presente (la casella è “vuota”). Ciò provoca un'interrupt da parte dell'MMU che indica che la pagina non è presente in memoria, che a sua volta richiama il *gestore della memoria virtuale* il quale avvia il processo di caricamento dalla memoria secondaria a quella principale.



Paginazione su richiesta

Per implementare la paginazione su richiesta, una copia dell'intero spazio di indirizzamento logico di un processo è mantenuta su un disco. L'area del disco utilizzata per memorizzare questa copia è chiamata spazio di swap del processo.

Durante l'inizializzazione di un processo, il gestore della memoria virtuale alloca lo spazio di swap per il processo e copia il suo codice e i suoi dati in tale spazio

Distinguiamo diversi tipi di **operazione**:

- **Page-in:** caricamento in memoria da disco della pagina del processo che fa riferimento ad un'istruzione o dato in una pagina non in memoria;
- **Page-out:** rimozione dalla memoria di una pagina e sua memorizzazione su disco. Notare che se tale pagina non è stata modificata dall'ultimo caricamento, questa non viene memorizzata.
- **Sostituzione della pagina:** caricamento di una pagina in un frame che conteneva un'altra pagina. È gestita da un algoritmo che combina le operazioni di page-out (se la pagina è stata modificata) e page-in.

Per gestire tali operazioni, il gestore ha bisogno di alcune informazioni.

Per tal motivo, ogni entrata della tabella delle pagine contiene delle **informazioni aggiuntive** su ciascuna pagina del processo:

Informazioni varie						
Bit di validità	Frame #	Prot info	Ref info	Modificato	Altre info	

Campo	Descrizione
Bit di validità	Indica se la pagina descritta dall'elemento attualmente è presente in memoria. Questo bit è anche chiamato bit di presenza.
Frame #	Indica quale frame di memoria è occupato dalla pagina.
Prot info	Indica le modalità di utilizzo del contenuto della pagina, se in scrittura, lettura o esecuzione.
Ref info	Fornisce informazioni riguardanti i riferimenti fatti alla pagina mentre era in memoria.
Modificato	Indica se la pagina è stata modificata mentre era in memoria, ovvero se è dirty. Questo campo è un singolo bit, detto dirty bit.
Altre info	Altre informazioni utili relative alla pagina, per esempio, la sua posizione nello spazio di swap.

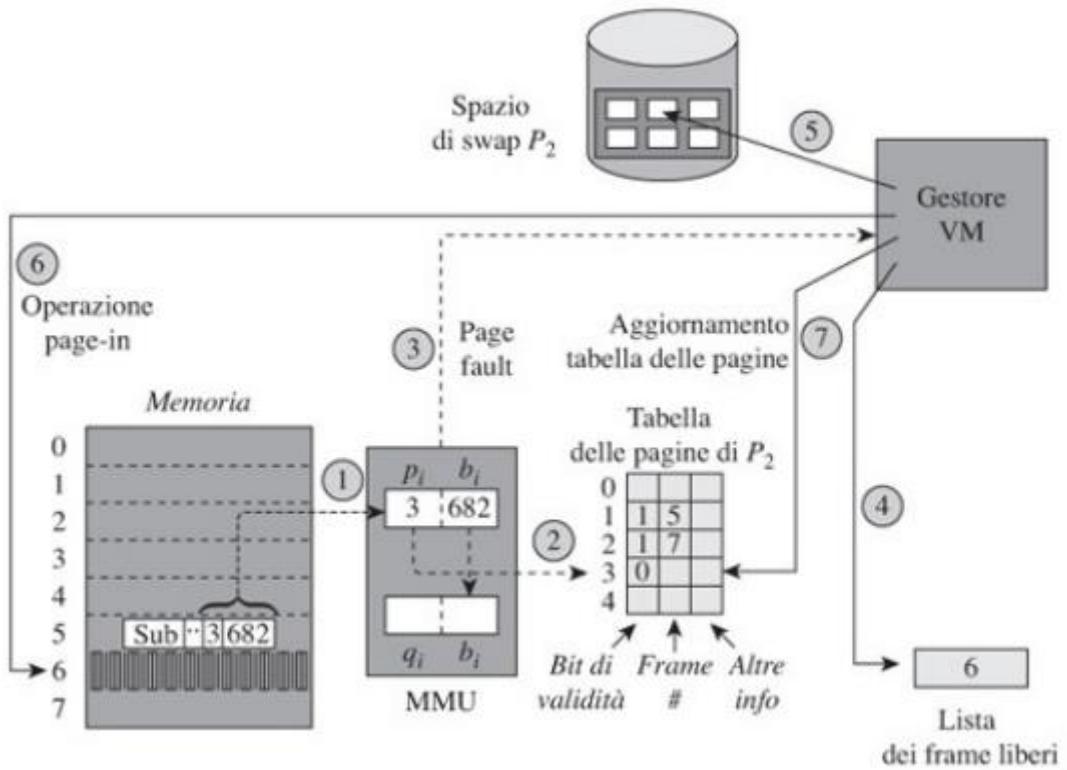
Paginazione su richiesta: FAULT DI PAGINA

Se la pagina che contiene l'indirizzo logico non è in memoria, la MMU genera un'interrupt di *page fault*.

Passo	Descrizione
1. Ricavare numero di pagina e numero di byte in una pagina	Un indirizzo logico è visto come una coppia (p_i, b_i) , dove b_i è dato dagli n_b bit meno significativi dell'indirizzo e p_i dato dagli n_p bit più significativi (Paragrafo 11.8).
2. Ricercare nella tabella delle pagine	p_i è usato per indicizzare la tabella delle pagine. Un page fault viene generato se il <i>bit di validità</i> dell'elemento della tabella delle pagine contiene uno 0, ossia se la pagina non è presente in memoria.
3. Formare l'indirizzo effettivo di memoria	Il campo <i>frame #</i> dell'elemento della tabella delle pagine contiene un numero di frame rappresentato con un numero a n_f bit. Se è concatenato con b_i si ottiene l'effettivo indirizzo di memoria del byte.

La MMU e il gestore della memoria virtuale interagiscono per decidere quando la pagina di un processo deve essere caricata in memoria.

Esempio di rappresentazione dei passi:



In questo esempio è generato un page fault perché il Bit di validità della pagina 3 è 0.
In tal caso possiamo distinguere ulteriori passi.

Passo 4/5: avvia operazione di I/O per caricare la pagina nel 3 nel frame 6.

Passo 6: viene effettuato il page-in, cioè all'interno del frame 6 è caricata la pagina 3.

Passo 7: viene aggiornata la tabella delle pagine, modificando il Bit di validità ad 1.

Con il *page fault*, la pagina richiesta è caricata in un frame di pagina libero.

Tuttavia se nessun frame è libero, il gestore della memoria virtuale esegue un'operazione di sostituzione della pagina. Quindi applico l'algoritmo di sostituzione della pagina ed avvio un page-out se la pagina è dirty (il bit modified è 1).

Le operazioni di page-in e page-out prendono il nome di *traffico di pagina*.

Tempo di accesso in memoria effettivo

Tale tempo di accesso è il tempo medio di accesso atteso da un processo, il quale dipende da due fattori:

- il tempo usato dalla MMU per la traduzione degli indirizzi;
- il tempo medio usato dal gestore della memoria virtuale nella gestione di un page fault.

Teniamo in considerazione la seguente **notazione**:

pr_1	probabilità che una pagina esista in memoria
t_{mem}	tempo di accesso alla memoria
t_{pfh}	overhead di tempo nella gestione del page fault

Da cui costruiamo la seguente **formula**:

$$\begin{aligned} \text{tempo di accesso effettivo alla memoria} &= pr_1 \times 2 \times t_{mem} \\ &+ (1 - pr_1) \times (t_{mem} + t_{pfh} + 2 \times t_{mem}) \end{aligned}$$

Se la pagina fosse in memoria basterebbe: $pr_1 \times 2 \times t_{mem}$.

pr_1 è anche detto *hit ratio*. moltiplichiamo t_{mem} per 2 perché accediamo due volte in memoria: accediamo sia alla tabella delle pagine che al frame.

Se la pagina non fosse in memoria dovremmo *aggiungere* alla quantità di prima la probabilità che la pagina NON sia in memoria (calcolato come $1 - pr_1$).

Per scoprire che la pagina non fosse in memoria avremmo sicuramente fatto un accesso alla tabella delle pagine, a cui aggiungiamo un tempo provocato dalla gestione del page fault e faremo poi inoltre altri due accessi alla memoria (accediamo alla tabella delle pagine e carichiamo la pagina nella memoria fisica).

Quindi moltiplicheremo la probabilità che la pagina NON sia in memoria con:

$t_{mem} + t_{pfh} + 2 \times t_{mem}$. Da cui otteniamo la formula finale.

L'EAT si può abbassare riducendo i page fault, e la strategia utilizzata dipende da SO a SO:

Paginazione su richiesta: SOSTITUZIONE DELLE PAGINE

La sostituzione di pagina diventa necessaria quando si verifica un page fault e non ci sono page frame liberi in memoria. Tuttavia si deve stare attenti a non rimuovere una pagina che potrebbe essere richiamata subito dopo.

Per tal motivo vengono utilizzati degli algoritmi che si basano sulla *località dei riferimenti*.

Il principio empirico della **località dei riferimenti** afferma che gli indirizzi logici usati dai processi, nel breve tempo, tendono a raggrupparsi in specifiche porzioni del loro spazio di indirizzamento logico.

Ciò avviene per **due ragioni**:

- l'esecuzione delle istruzioni è generalmente sequenziale, e solo una più piccola parte di tali istruzioni è di "salto";
- i processi tendono ad eseguire operazioni simili sugli elementi di dati non scalari (es. array).

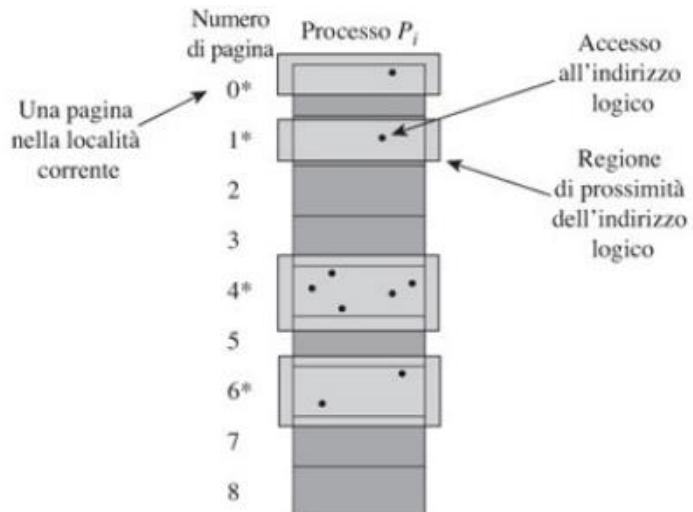
Da qui osserviamo che le istruzioni e i riferimenti ai dati tendono ad essere in prossimità di un'istruzione precedente o di un dato referenziato, e per tal motivo è stato introdotto il concetto di *località corrente*.

La **località corrente** di un processo è l'insieme di pagine referenziate in poche istruzioni precedenti; questa informazione consente di abbassare il numero di page fault.

I page fault sono tuttavia sempre possibili. Infatti se prendiamo in considerazione una **Regione di prossimità** di un indirizzo logico a_i , la quale è definita come l'insieme degli indirizzi logici in prossimità dell'indirizzo a_i , allora possono verificarsi page fault se:

- la regione di prossimità non entra in una sola pagina;
- un'istruzione o dato referenziato da un processo può non essere in prossimità dei riferimenti, per un fenomeno detto **shift della località** di un processo.

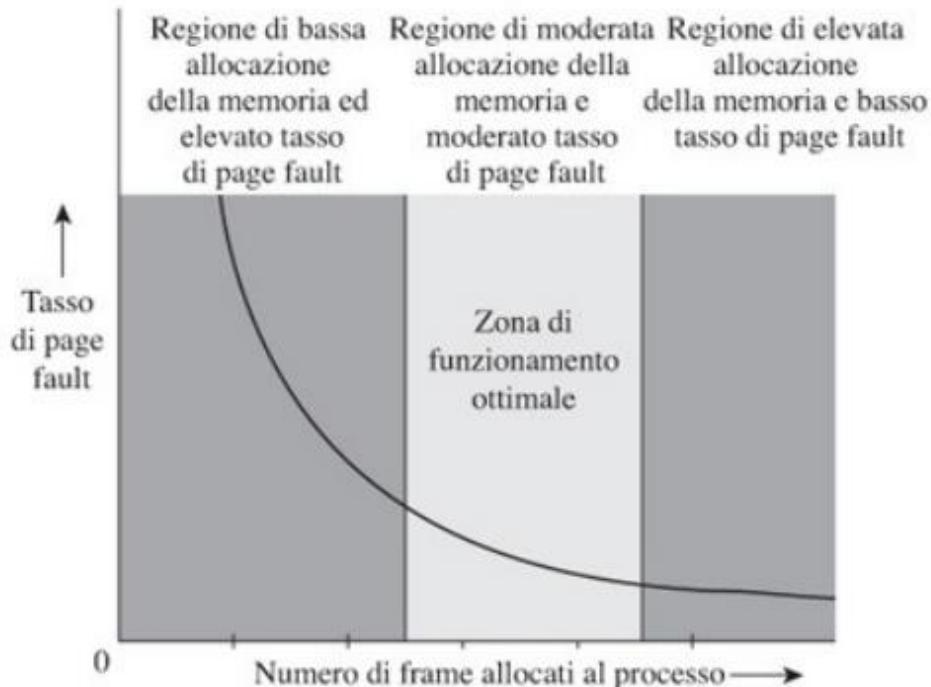
L'algoritmo di sostituzione, in pratica, deve evitare di sostituire le pagine che fanno parte della località corrente (*), però è anche possibile rimuovere una pagina che non fa parte della località corrente ma che è coinvolta in parte dalla regione di prossimità (es. 3). Cioè è probabile che ci siano dei riferimenti a quella pagina che potrebbero essere richiamati subito dopo, ma siccome la pagina potrebbe essere stata stata rimossa si genera un *page fault*.



ALLOCAZIONE DELLA MEMORIA AD UN PROCESSO

La percentuale dei page di un processo varia in funzione della quantità di memoria a essi allocata. Maggiore è la quantità di memoria allocata (num. di frame), minore sarà la probabilità di un page fault.

Ovviamente più memoria alloco ad un processo, più ne levo ad altri (riduco il grado di multiprogrammazione). Per tal motivo si dovrebbe scegliere un giusto compromesso fra memoria allocata e probabilità di page fault, detta *zona di funzionamento ottimale*.



La zona a sinistra potrebbe comportare un'elevata successione di operazioni di I/O, che comporta una CPU in stato idle. Si parla di *Thrashing*.

Il **Thrashing** è una condizione in cui coincidono un elevato traffico di pagina e bassa efficienza della CPU.

Dimensione della pagina ottimale

La dimensione della pagina è definita dall'hardware, ed impatta diverse aspetti:

- numero di bit richiesti per rappresentare il numero di byte in una pagina;
- spreco di memoria causato da frammentazione interna;
- dimensione della tabella delle pagine per un processo;
- i tassi di page fault quando una quantità fissa di memoria è allocata ad un processo.

L'uso di dimensioni di pagina maggiori rispetto al valore ottimale implica page fault maggiori, ed anche in questo caso si cerca un compromesso tra costi HW e operazioni efficienti.

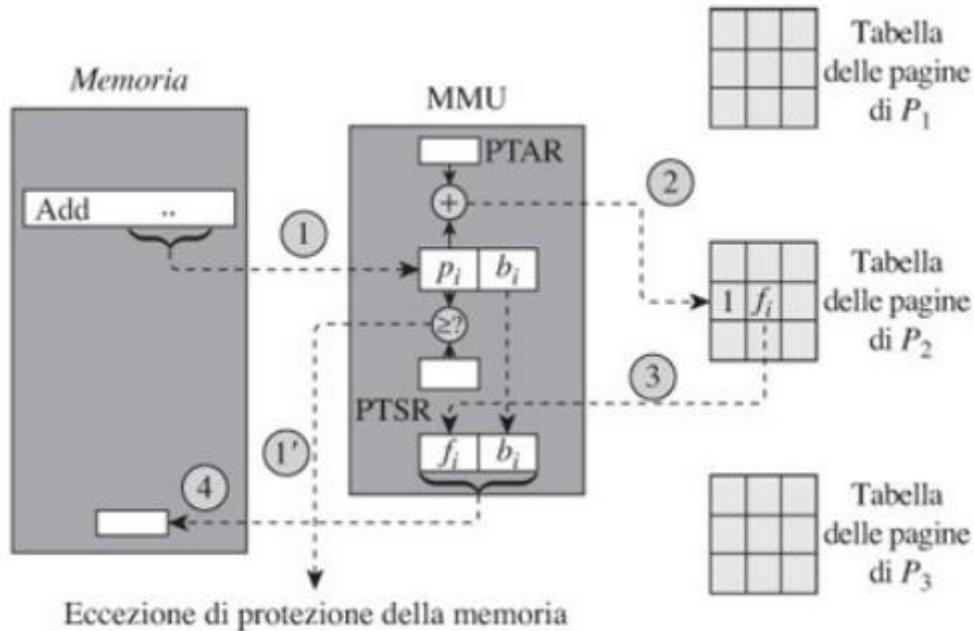
HARDWARE DI PAGINAZIONE

In un sistema multiprogrammato si hanno molti processi in memoria, dove per ogni processo si ha una tabella delle pagine. Per cui è necessario avere un supporto per ciascun processo per identificare in modo efficiente ciascuna tabella delle pagine.

La MMU contiene un apposito registro detto PTAR (*Registro indirizzo della tabella delle pagine*) il quale punta all'inizio di una tabella delle pagine.

Per un indirizzo logico (p_i, b_i), la MMU calcola $\text{PTAR} + p_i \times l_{\text{PT_entry}}$ per ottenere l'indirizzo della entry della pagina p_i nella tabella delle pagine, dove $l_{\text{PT_entry}}$ è la lunghezza di una entry della tabella delle pagine e PTAR denota il contenuto della PTAR. La PTAR deve essere caricata con l'indirizzo corretto quando viene schedulato un processo.

Per facilitare questo, il kernel può memorizzare l'indirizzo della tabella delle pagine di ogni processo nel suo blocco di controllo del processo (PCB).



Un HW di paginazione svolge anche una serie di funzioni, quali: Protezione della memoria, Efficiente traduzione degli indirizzi e Supporto per la sostituzione delle pagine.

Protezione della memoria

Il paging hardware su occupa di assicurare che un processo accede solo a quelle aree di memoria per esso allocate.

È generata una **violazione** della protezione della memoria se:

- un processo cerca di accedere ad una pagina che non esiste;
- il processo viola i suoi privilegi di accesso (alla pagina).

Esso è **implementato** attraverso:

- il *registro dimensione della tabella delle pagine* (PTSR) della MMU, con il quale si controlla che un processo acceda ad una sua pagina effettiva e non ad una pagina che non esiste all'interno del processo. Quindi si confronta il numero di pagina con la dimensione del PTSR, e se fosse maggiore allora la pagina non esiste e viene lanciata un'interrupt.
- il campo *Prot info* dell'entrata della pagina nella tabella delle pagine, che contiene i diritti di accesso del processo alla pagina.
I permessi sono codificati come bit, e se non combaciassero verrebbe lanciata un'interrupt.

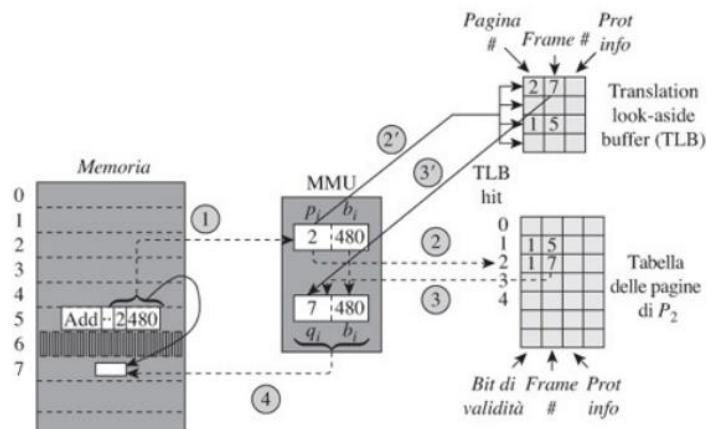
Traduzione indirizzo e generazione page fault

Durante la traduzione degli indirizzi si utilizza un ciclo di memoria perché la tabella delle pagine è memorizzata in memoria. Si può velocizzare il tutto attraverso il TLB.

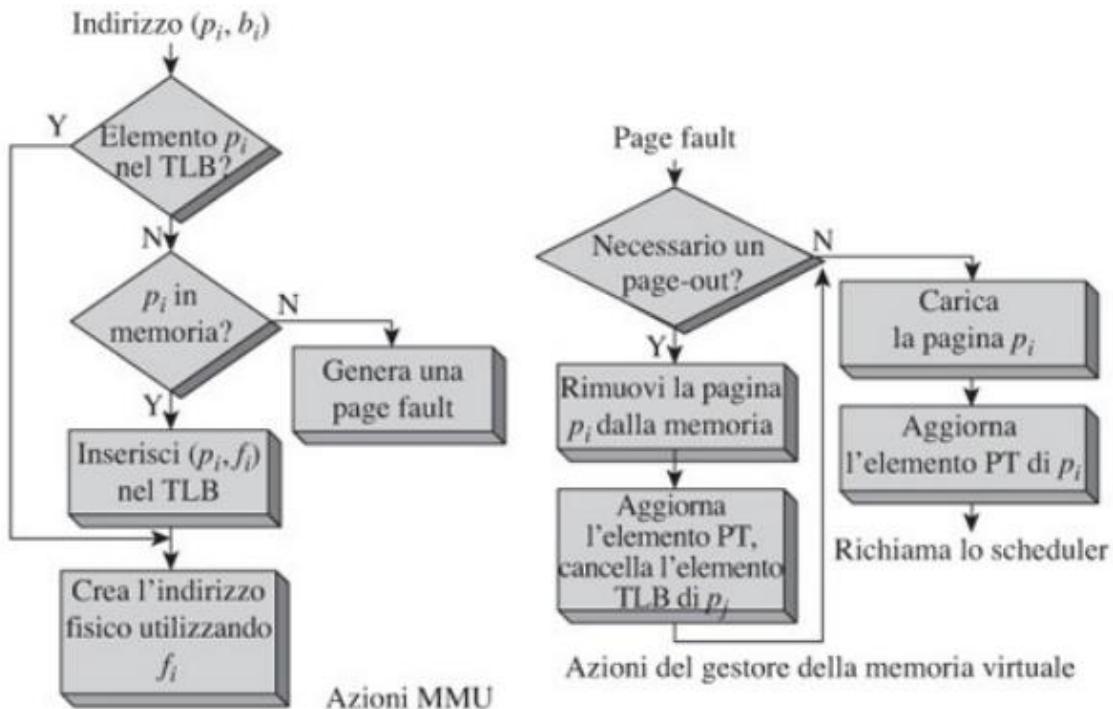
Il *Translation look-aside buffer* (TLB): memoria associativa piccola e veloce usata per accelerare la traduzione dell'indirizzo.

In pratica si cerca di fare in modo che le informazioni sulle pagine, per la traduzione, piuttosto che reperirle nella tabella delle pagine che si trova in memoria centrale, siano presenti in questo piccolo buffer che è molto più veloce.

Se si usa il TLB, si verifica innanzitutto se la pagina è presente in TLB. In caso affermativo accederò direttamente nel TLB senza accedere alla tabella delle pagine, altrimenti dovrò anche accedere a tale tabella.



■ Schema di traduzione degli indirizzi (p_i, b_i):



I TLB possono essere gestiti sia in HW (a sinistra) che in SW (a destra).

Sebbene la gestione in SW sia un po' più lenta è anche più flessibile, così facendo la MMU esegue la traduzione degli indirizzi esclusivamente interagendo con il TLB, mentre la parte della gestione delle pagine è eseguita dal gestore.

In questo modo il gestore può utilizzare diverse organizzazioni delle tabelle delle pagine per ottimizzare l'uso della memoria.

Inoltre, l'uso del TLB può minare la protezione se la MMU esegue la traduzione con indirizzi con entrate del TLB create durante l'esecuzione di altri processi, per cui:

- ogni entrata del TLB può contenere l'id del processo in esecuzione nel momento della creazione dell'entrata. Così facendo, la MMU evita di usarla quando altri processi sono in esecuzione;
- oppure, il kernel deve scaricare (flush) il TLB mentre esegue la commutazione tra processi.

■ Tempo di accesso in memoria effettivo (con TLB)

Data la seguente **notazione**:

- pr_1 probabilità che una pagina esista in memoria;
 pr_2 probabilità che una entry di pagina esista nel TLB;
 t_{mem} tempo di accesso in memoria;
 t_{TLB} tempo di accesso al TLB;
 t_{pfh} overhead di tempo per la gestione dei page fault.

Possiamo costruire la **formula**:

Tempo effettivo di accesso alla memoria =

$$pr_2 \times (t_{TLB} + t_{mem}) + (pr_1 - pr_2) \times (t_{TLB} + 2 \times t_{mem}) \\ + (1 - pr_1) \times (t_{TLB} + t_{mem} + t_{pfh} + t_{TLB} + 2 \times t_{mem})$$

pr_2 è anche detto *TLB hit ratio*. Se la pagina che deve essere tradotta si trova nel TLB basterebbe la formula: $pr_2 \times (t_{TLB} + t_{mem})$.

Cioè la TLB hit ratio viene moltiplicata per la somma fra il tempo per accedere al TLB e il tempo per accedere alla memoria.

Se invece la pagina non si trova all'interno del TLB ma si trova nella tabella delle pagine ci si potrebbe fermare al secondo prodotto.

In particolare, $(pr_1 - pr_2)$ fa riferimento al caso in cui la pagina non sta nel TLB ma sta nella memoria, il quale è moltiplicato per il tempo per accedere alla memoria + 2 volte il tempo per accedere alla memoria.

Se invece la pagina non si trova né all'interno del TLB e né nella tabella delle pagine si deve applicare l'intera formula.

$(1 - pr_1)$ rappresenta la probabilità che la pagina NON sia in memoria, la quale è moltiplicata per tutto il tempo di accesso alle memorie: il tempo per accedere al TLB, il tempo per accesso alla memoria, l'overhead generato dal page fault + 2 volte il tempo per accedere alla memoria (perché aggiorno l'entrata della tabella delle pagine).

Sono usati alcuni **meccanismi** per migliorare le prestazioni:

- le entrate wired TLB per le pagine del kernel (mai sostituite);
- le superpagine.

Supporto per la sostituzione delle pagine

Si occupa di collezionare informazioni relative ai riferimenti fatti alle pagine. Il gestore della memoria virtuale utilizza questa informazione per decidere quale pagina sostituire quando si verifica un page fault, così da minimizzare i page fault e il numero di operazioni page-in e page-out.

Le **informazioni** sono:

- l'istante in cui una pagina è stata usata l'ultima volta.
Tuttavia conservare un istante di tempo è costoso da un punto di vista di numero di bit, per tal motivo si preferisce usare un singolo bit di riferimento;
- se una pagina è dirty.
Si usa il bit modified per capire se la pagina è stata modificata.

ORGANIZZAZIONE DELLE TABELLE DELLE PAGINE IN PRATICA

Un processo con un ampio spazio di indirizzamento richiede un'ampia tabella delle pagine. Di conseguenza, il gestore della memoria virtuale deve allocare un grosso quantitativo di memoria per ogni tabella delle pagine.

La dimensioni di tale tabelle sarebbe così grande da non risultare fattibile. Per tale motivo si possono adottare due **soluzioni**: *tabella delle pagine invertita* e *tabella delle pagine multilivello*.

In entrambi gli approcci, il TLB viene usato per ridurre il numero di riferimenti di memoria necessari per eseguire la traduzione degli indirizzi.

Tabella delle pagine invertita

Invece di mantenere informazioni sulle pagine di ogni frame, mantiene informazioni sui frame di ogni pagina.

In tal modo la dimensione della tabella non dipende più dalla dimensione dei processi presenti e dalla dimensioni di ciascuna pagina, ma solo dalla dimensione della memoria della macchina.

Contiene coppie della forma (id programma, #pagina).

Il **limite** di questa tabella è che le informazioni su una pagina devono essere cercate. La ricerca viene effettuata con tabelle *hash*, così da velocizzarla.

Tabella delle pagine multilivello

La tabella delle pagine di un processo è a sua volta paginata; non è, perciò, necessario che l'intera tabella delle pagine sia presente sempre in memoria.

L'accesso ad una tabella di livello inferiore deve essere fatto da una tabella di livello superiore.

LEZIONE 22 – MEMORIA VIRTUALE (2)

POLITICHE DI SOSTITUZIONE DELLE PAGINE

Una politica di sostituzione delle pagine dovrebbe sostituire solo quelle pagine che probabilmente non saranno riferite nell'immediato. Ne distinguiamo 3:

- Politica di sostituzione delle pagine ottimale. (minimizza il num tot di page fault)
- Politica di sostituzione delle pagine First-In, First-Out (FIFO).
- Politica di sostituzione delle pagine Least Recently Used (LRU)

Per l'analisi di queste politiche di sostituzione di pagina, ci basiamo sul concetto di *stringa dei riferimenti alle pagine*.

Una **stringa dei riferimenti alle pagine** di un processo è una sequenza di pagine a cui un processo ha fatto accesso durante la sua esecuzione.

Può essere costruita monitorando l'esecuzione di un processo, e formando una sequenza di numeri di pagina che appaiono negli indirizzi logici generati da esso.

Si associa ad ogni stringa di riferimento pagina una *stringa dei riferimenti temporale* t_1, t_2, t_3, \dots . In questo modo, assumiamo che il k -esimo riferimento di pagina nella stringa dei riferimenti alle pagine dovrà verificarsi all'istante di tempo t_k .

Esempio 12.5 Stringa dei riferimenti alle pagine

Un computer supporta istruzioni di lunghezza 4 byte, e usa una dimensione di pagina di 1 KB. Esegue il seguente programma, privo di senso, in cui i simboli *A* e *B* sono nelle pagine 2 e 5, rispettivamente:

	START	2040
	READ	B
LOOP	MOVER	AREG, A
	SUB	AREG, B
	BC	LT, LOOP
	...	
	STOP	
A	DS	2500
B	DS	1
	END	

La stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti per il processo sono le seguenti:

stringa dei riferimenti alle pagine 1, 5, 1, 2, 2, 5, 2, 1, ...
stringa dei tempi dei riferimenti $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, \dots$

L'indirizzo logico della prima istruzione è 2040, e quindi si trova nella pagina 1. Il primo riferimento di pagina nella stringa, perciò, è 1. Siamo all'istante di tempo t_1 . *B*, l'operando dell'istruzione, è situato nella pagina 5, e quindi il secondo riferimento di pagina nella stringa è 5, all'istante t_2 . La prossima istruzione è situata nella pagina 1 e si riferisce ad *A*, che si trova nella pagina 2, e quindi i due successivi riferimenti di pagina sono alle pagine 1 e 2. Le successive due istruzioni sono situate alla pagina 2, e l'istruzione con la label *LOOP* si trova nella pagina 1. Perciò, se il valore dell'input *B* all'istruzione *READ* è maggiore del valore *A*, i successivi quattro riferimenti di pagina sarebbero alle pagine 2, 5, 2 e 1, rispettivamente; altrimenti, i successivi quattro riferimenti di pagina sarebbero alle pagine 2, 5, 2 e 2, rispettivamente.

SOSTITUZIONE DELLE PAGINE OTTIMALE

La sostituzione ottimale consiste nel prendere decisioni di sostituzione in modo tale che il numero totale di page fault, durante l'esecuzione di un processo, sia il **più piccolo possibile**; ossia, nessun'altra sequenza di sostituzione di pagine può determinare un numero inferiore di page fault.

Per realizzarlo, ad ogni page fault, la strategia di sostituzione dovrebbe considerare tutte le possibili decisioni alternative, analizzare le implicazioni per i page fault futuri e selezionare la miglior alternativa.

Ovviamente tale soluzione non è praticabile in quanto il gestore non ha conoscenza del comportamento futuro di un processo.

Tuttavia Belady dimostrò che tale soluzione è equivalente ad applicare tale **regola**: al verificarsi di un page fault, sostituire la pagina il cui prossimo riferimento è il più lontano nella stringa dei riferimenti di pagina.

SOSTITUZIONE FIFO

La sostituzione FIFO, ad ogni page fault, sostituisce la pagina caricata in memoria prima di ogni altra pagina del processo.

Per semplificare il lavoro, il gestore memorizza nel campo *ref info* l'istante di caricamento di una pagina.

SOSTITUZIONE LRU

La sostituzione LRU usa la legge dei riferimenti. Ad ogni page fault, è sostituita la pagina usata meno recentemente.

L'entrata della tabella delle pagine registra il tempo dell'ultimo riferimento alla pagina. Questa informazione è inizializzata quando una pagina è caricata, ed è aggiornata ogni volta che la pagina è riferita.

Esempio sostituzione ottimale, FIFO, LRU

Consideriamo le seguenti stringhe di riferimento pagina e stringhe di riferimento temporale per un processo P

Stringa riferimento pagina	0, 1, 0, 2, 0, 1, 2, ...
Stringa riferimento temporale	t ₁ , t ₂ , t ₃ , t ₄ , t ₅ , t ₆ , t ₇ , ...

Vediamo il funzionamento delle diverse strategie di sostituzione pagine con *alloc* = 2, dove alloc rappresenta il numero di frame di pagina allocati al processo P.

		Ottimale		FIFO		LRU	
Istante	Rif. pagina	Bit di validità info	Ref	Bit di validità info	Ref	Bit di validità info	Ref
t_1	0	0 1 1 0 2 0	-	0 1 t_1 1 0 2 0	-	0 1 t_1 1 0 2 0	-
t_2	1	0 1 1 1 2 0	-	0 1 t_1 1 1 t_2 2 0	-	0 1 t_1 1 1 t_2 2 0	-
t_3	0	0 1 1 1 2 0	-	0 1 t_1 1 1 t_2 2 0	-	0 1 t_3 1 1 t_2 2 0	-
t_4	2	0 1 1 0 2 1	Sostituisci 1 con 2	0 0 1 1 t_2 2 1 t_4	Sostituisci 0 con 2	0 1 t_3 1 0 2 1 t_4	Sostituisci 1 con 2
t_5	0	0 1 1 0 2 1	-	0 1 t_5 1 0 2 1 t_4	Sostituisci 1 con 0	0 1 t_5 1 0 2 1 t_4	-
t_6	1	0 0 1 1 2 1	Sostituisci 0 con 1	0 1 t_5 1 1 t_6 2 0	Sostituisci 2 con 1	0 1 t_5 1 1 t_6 2 0	Sostituisci 2 con 1
t_7	2	0 0 1 1 2 1	-	0 0 1 1 t_6 2 1 t_7	Sostituisci 0 con 2	0 0 1 1 t_6 2 1 t_7	Sostituisci 0 con 2

Il caso *ottimale* carica in memoria la pagina referenziata (se possibile) e imposta il BV ad 1, altrimenti deve effettuare la sostituzione con la pagina caricata più tardi.
Se la pagina riferita è già in memoria, non succede nulla.

All'istante t_1 viene referenziata la pagina 0, tale pagina è caricata in memoria e il BV diventa 1.
All'istante $t_2 \rightarrow 1$, anche questa è caricata in memoria e il BV diventa 1. All'istante $t_3 \rightarrow 0$, ma è già caricata in memoria quindi non accade nulla.
All'istante $t_4 \rightarrow 2$, ma abbiamo già due pagine in memoria quindi deve avvenire una sostituzione. Sostituiamo la 1 perché è quella riferita più tardi. e così via...

Il caso *FIFO* marca temporalmente ogni pagina la prima volta che è stata caricata in memoria. Se deve effettuare una sostituzione, sostituirà la pagina che è stata caricata per prima (informazione in *ref info*, quella col riferimento temporale minore).

Il caso *LRU* marca temporalmente ogni pagina ogni volta che questa è referenziata. Se deve fare una sostituzione, sostituirà la pagina usata meno recentemente (quella col riferimento temporale minore).

(numero page fault \rightarrow ottimale = 4, FIFO = 6, LRU = 5)

Confronto ottimale, FIFO, LRU

In generale, si ha che la sostituzione ottimale abbia il minore numero di page fault, seguito dalla LRU, seguito dalla FIFO. Infatti

La FIFO seleziona la pagina da sostituire in base al caricamento in memoria più vecchio. Si tratta però di una strategia non desiderabile, perché non è detto che la pagina caricata da più tempo non venga più referenziata, anzi potrebbe anche capitare che venga referenziata spesso dal suo primo caricamento. Ciò comporta un maggior numero di page fault.

La LRU è una miglior strategia di quella FIFO proprio perché fa riferimento alla pagina usata meno recentemente, ovvero a quella *referenziata* più tardi.

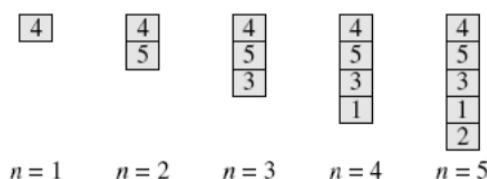
PROPRIETÀ DELLO STACK

Per ottenere caratteristiche desiderabili di page fault, i page fault non dovrebbero aumentare quando si incrementa l'allocazione della memoria.

Perché ciò avvenga, la strategia adottata deve godere della *proprietà dello stack*.

Una politica di sostituzione di pagina possiede la **proprietà dello stack** se $\{pi\}_n^k$ è incluso in $\{pi\}_m^k$ per tutti gli n, m tali che $n < m$

dove $\{pi\}_n^k$ indica l'insieme delle pagine in memoria al tempo t_k^+ se $alloc_i = n$ durante l'intera attività del processo P_i (t_k^+ implica l'istante dopo t_k ma prima di t_{k+1})



$\{pi\}_n^k$ per differenti n per una politica di sostituzione di pagina che **implementa** la proprietà dello stack. L'insieme delle pagine allocato quando $n=1$ è compreso nell'insieme delle pagine quando $n=2$, e l'insieme delle pagine quando $n=2$ è compreso nell'insieme delle pagine quando $n=3$, e così via...

dimostrazione

Consideriamo due esecuzioni del processo P_i , con $alloc_i = n$ e $alloc_i = m$ rispettivamente, dove $n < m$.

Se una politica ha la proprietà dello stack, significa che fissato un istante temporale k tutte le pagine presenti quando $alloc_i = n$ devono essere presenti anche quando $alloc_i = m$.

Se questo è vero, è anche vero che quando eseguo un processo con $alloc_i = m$ ci saranno in più altre $m-n$ pagine.

Se una di tali pagine sarà riferita in pochi riferimenti successivi di P_i , il page fault si verifica se $alloc_i = n$, ma non se $alloc_i = m$.

Quindi il page fault è più elevato se $alloc_i = n$ rispetto ad $alloc_i = m$.

PROBLEMI CON POLITICA FIFO

Quando si osserva l'algoritmo FIFO possiamo notare che il numero di page fault aumenta quando aumenta l'allocazione della memoria per il processo. Questo comportamento anomalo fu riportato per primo da Belady ed è perciò noto come *anomalia di Belady*.

Il gestore della memoria virtuale non può usare la sostituzione di pagina FIFO perché incrementando l'allocazione per un processo può aumentare la frequenza dei page fault del processo. Questa caratteristica renderebbe difficile affrontare il thrashing nel sistema.

Tuttavia, quando viene usata la sostituzione di pagina LRU, il numero di page fault è una funzione non crescente di alloc. Di conseguenza è possibile combattere il thrashing incrementando il valore di alloc per ogni processo.

POLITICHE DI SOSTITUZIONE PAGINA IN PRATICA

Il gestore della memoria virtuale mantiene una *lista di frame liberi* e prova a tenere pochi frame in questa lista a ogni istante. Il gestore della memoria virtuale consiste di due thread **demoni**, così chiamati perché sono sempre in esecuzione.

Il gestore dei *frame liberi* viene attivato ogni qualvolta la *lista di frame liberi* contiene un numero di frame al di sotto di una certa soglia definita dal gestore.

Il gestore va quindi a scandire le pagine dei processi presenti in memoria in modo tale da rimuovere qualche pagina e inserirla nella lista.

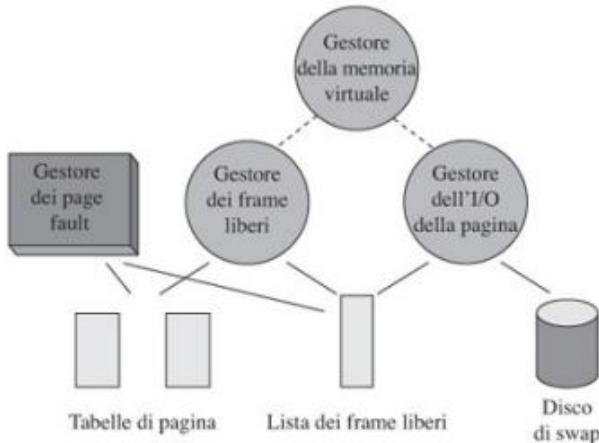
Il gestore segna anche all'interno di questa lista di frame se quel frame contiene una pagina che era stata modificata di recente (aggiorna il dirty bit), ed inoltre modifica il bit di validità.

Il gestore dell'I/O di pagina esegue le operazioni di page-in/page-out.

In particolare, se una pagina ha dirty bit ad 1, il gestore sposta la pagina su disco e ne imposta il dirty bit a 0.

Il gestore del *Page fault* si attiva ogni qualvolta c'è un page fault. Verifica se la pagina che ha richiesto di essere caricata in memoria si trova nella *lista di frame liberi*, e in tal caso non c'è bisogno di fare un page-in (perché di fatto la pagina è ancora in memoria) ma ne modifica solo il bit di validità, e lo rimuove dalla lista.

In caso contrario, si deve effettuare una vera operazione di page-in, ovvero il caricamento della pagina dal disco.



Nella realtà, la sostituzione **FIFO** è scartata a prescindere perché non implementa la *proprietà dello stack*.

Tuttavia, anche la sostituzione **LRU** non è fattibile in quanto i computer non forniscono sufficienti bit nel campo *ref info* per memorizzare l'istante dell'ultimo riferimento.

La maggior parte dei computer forniscono un singolo bit di riferimento, il quale è utilizzato dalle politiche *Not Recently Used* (NRU).

La strategia NRU più semplice prevede di inizializzare tale bit a 0 quando la pagina è caricata per la prima volta in memoria. Nel momento in cui si deve sostituire una pagina, va alla ricerca di pagine con *ref info* 0. Se tutte le pagine hanno tale bit ad 1, resetta tutti i bit a 0 e ne sceglie una da sostituire.

Si tratta di una strategia semplice da implementare ma non discrimina bene (non è detto che scelga la pagina che è stata referenziata meno recentemente). Per tal motivo esistono delle *varianti*, detti *algoritmi di clock*, che forniscono una miglior discriminazione di pagina in quanto resettando i bit di riferimento *periodicamente*.

ALGORITMI DI CLOCK

Gli algoritmi di clock forniscono una miglior discriminazione di pagina in quanto resettano i bit di riferimento *periodicamente*.

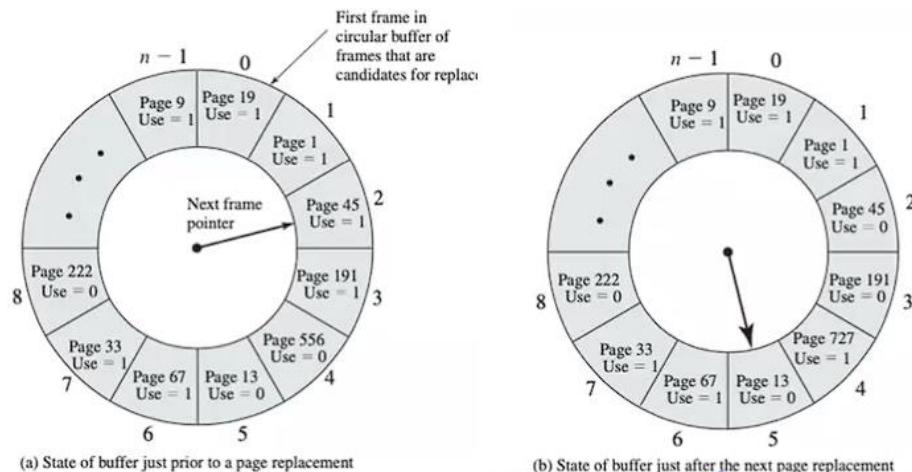
Le pagine di tutti i processi in memoria sono immessi in una lista circolare e sono usati dei puntatori che si spostano sulle pagine ripetutamente.

È esaminata la pagina puntata da un puntatore, e su questa viene intrapresa un'azione. Il puntatore è aggiornato per puntare alla prossima pagina.

Clock ad una Lancetta

Una **scansione** consiste di due passi su tutte le pagine:

1. il gestore della memoria virtuale resetta il bit di riferimento per la pagina puntata dal puntatore;
2. trova tutte le pagine i cui bit di riferimento sono 0 e le pone nella *free list*.



Clock a due lancette

Nel clock a due lancette sono utilizzati due puntatori:

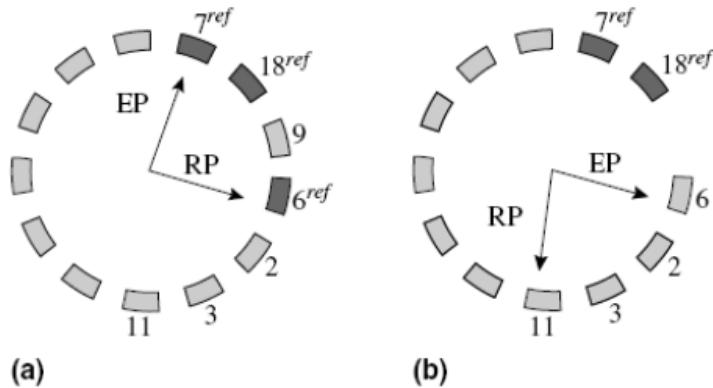
- **Puntatore di reset (RP)**: usato per resettare i bit di riferimento;
- **Puntatore di controllo (EP)**: usato per controllare i bit di riferimento.

Una volta esaminato un frame, i due puntatori sono incrementati simultaneamente. Il frame di pagina a cui punta EP è aggiunto alla lista dei frame liberi se il suo bit di riferimento è 0, altrimenti rimane inalterato.

Il *ref bit* del frame puntato da RP è impostato a 0.

La distanza fra i due puntatori è **fissata** dall'algoritmo, e in particolare:

- se i puntatori sono **vicini**, solo le pagine usate di recente resteranno in memoria;
- se i puntatori sono **distanti**, solo le pagine che non sono state usate da tanto tempo sono rimosse.



notare che quando EP arriva a 9, essendo che il suo bit di riferimento è 0, viene aggiunto alla lista dei frame liberi. **nb:** le lancette si muovono di una “posizione” alla volta.

In breve:

Viste le strategie, abbiamo visto che LRU è la migliore ma non è implementabile, in quanto è troppo costoso in termini di bit tenere traccia dell'istante temporale.

Si usa invece la variante RNU che usa un solo bit, dove le strategie più diffuse sono quelle di *clock* (ad una o a due lancette).

Tuttavia non sappiamo ancora il valore di alloci, ovvero quanti frame allocare per cercare di ridurre i page fault (fondamentale per sapere se ci troviamo nella zona desiderabile).

CONTROLLARE L'ALLOCAZIONE DI MEMORIA AD UN PROCESSO

Al processo P_i sono allocate alloct frame di pagina. Posso usare **due strategie**:

- **Allocazione di memoria fissa:** fissa alloc in modo statico. Quando si verifica un page fault in un processo, viene sostituita una delle sue pagine. Questo approccio è detto *sostituzione di pagina locale*.
Il problema di questa soluzione è che potrei allocare troppa o troppo poca memoria.
- **Allocazione di memoria variabile:** usa sostituzioni di pagina globali e/o locali.
Nel caso di *sostituzione globale*, quando si verifica un page fault, tutte le pagine di tutti i processi che sono presenti in memoria possono essere prese in considerazione per la sostituzione.
Nel caso di *sostituzione locale*, il gestore periodicamente determina il valore corretto di alloc per un processo.

La sostituzione *locale* si implementa attraverso il modello *working set*.

WORKING SET

Il **working set** è l'insieme di pagine di un processo che sono state referenziate nelle precedenti Δ istruzioni del processo, dove Δ è un parametro del sistema.

Notazione:

$WS_i(t, \Delta)$	working set per il processo $proc_i$ all'istante t con dimensione della finestra Δ ;
$WSS_i(t, \Delta)$	dimensione del working set $WS_i(t, \Delta)$, ossia il numero di pagine in $WS_i(t, \Delta)$.

Notare che $WSS_i(t, \Delta) \leq \Delta$ perché una pagina può essere riferita più di una volta in una finestra del working set.

Un *allocatore* di memoria basato su working set o *mantiene* l'intero working set in memoria oppure *sospende* il processo. Quindi si può avere che, al tempo t per il processo P_i , rispettivamente $alloc_i = WSS_i$ oppure $alloc_i = 0$.

Tale strategia assicura buoni hit ratio in memoria, quindi preserva la *località dei riferimenti*, e così facendo contiene il numero di page fault e di conseguenza previene il thrashing.

Gradi di multiprogrammazione

L'allocatore del working set varia il grado di multiprogrammazione sulla base delle dimensioni dei working set dei processi.

Per esempio, se $\{P_k\}$ è l'insieme dei processi in memoria,

- si decide di abbassare il grado di multiprogrammazione se “la somma di tutti i k processi dei working set è maggiore del numero di frame in memoria fisica”

$$\sum_k WSS_k > \#frame$$

- si decide di incrementare il grado di multiprogrammazione se $\sum_k WSS_k < \#frame$ ed esiste P_g tale che “il suo working set ha una dimensione minore del numero di frame in memoria fisica – la somma di tutti i k processi dei working set”:

$$WSS_g \leq (\#frame - \sum_k WSS_k)$$

Il gestore della memoria virtuale mantiene alloc_i e WSS_i per ogni P_i.

Per **abbassare** il grado di multiprogrammazione il gestore sceglie un processo, P_i, da sospendere. Poi esegue una page-out per ogni pagina modificata da P_i e cambia lo stato dei frame a libero.

Per il processo che stiamo sospendendo: alloc_i è impostato a 0, mentre WSS_i rimane inalterato.

Per **incrementare** il grado di multiprogrammazione, si ripristina P_i e si pone alloc_i = WSS_i.

Poi carica la pagina di P_i che contiene la prossima istruzione da eseguire, le altre pagine sono caricate in corrispondenza dei page fault.

Alternativamente, si caricano tutte le pagine di WS_i, ma è possibile che si verifichi ridondanza dei caricamenti.

Implementazione

È costoso determinare WS_i(t, Δ) e alloc_i ad ogni istante t. Per tal motivo viene determinato a *intervalli regolari*.

Gli insiemi determinati alla fine di un intervallo sono usati per decidere i valori di alloc da usare per il prossimo intervallo.

es.

60 frame liberi da allocare a
P₁, P₂, P₃ e P₄

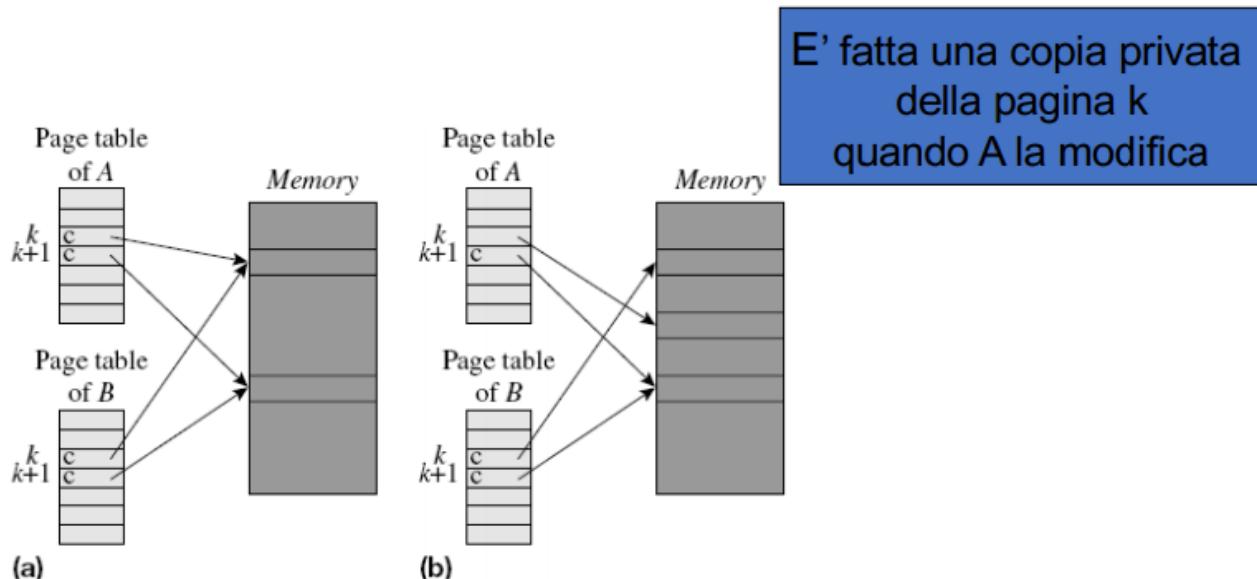
Process	t100		t200		t300		t400	
	WSS	alloc	WSS	alloc	WSS	alloc	WSS	alloc
P ₁	14	14	12	12	14	14	13	13
P ₂	20	20	24	24	11	11	25	25
P ₃	18	18	19	19	20	20	18	18
P ₄	10	0	10	0	10	10	12	0

COPY-ON-WRITE

La funzionalità copy-on-write è utilizzata per risparmiare memoria quando i dati nelle pagine condivise potrebbero essere modificati ma i valori modificati sono riservati a un processo.

Quando viene generato un processo figlio, le pagine del processo padre non vengono copiate. Inizialmente padre e figlio condividono la stessa copia in memoria, però viene utilizzata una flag *copy-on-write*.

Finché tali pagine non sono toccate, queste sono condivise fra i due processi, ma quando il processo figlio riferenzia un “qualcosa” del processo padre, la pagina viene effettivamente copiata. Entrambi i processi avranno dunque una *copia privata* della suddetta pagina.



LEZIONE 23 – FILE SYSTEM

INTRODUZIONE FILE SYSTEM e IOCS

Un **File System** indica un meccanismo con il quale i file sono posizionati e organizzati su dispositivi informatici utilizzati per l'archiviazione dei dati, o su dispositivi remoti tramite protocolli di rete.

Il File System vede un file come una collezione di dati di proprietà dell'utente, condiviso da un insieme di utenti autorizzati e memorizzato in modo affidabile per lungo tempo.

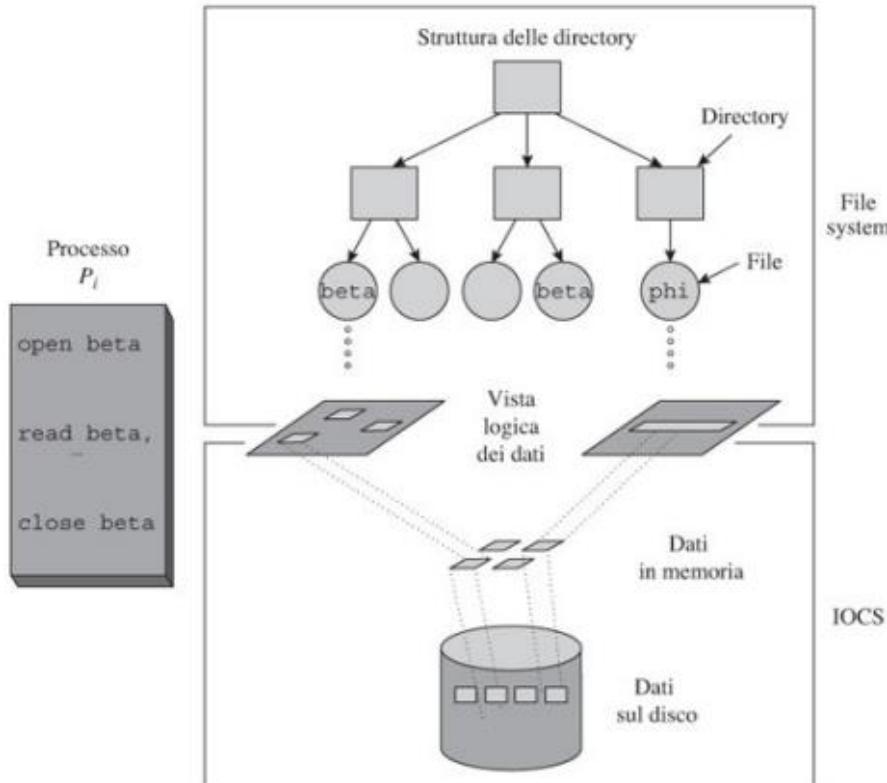
L'**IOCS** gestisce le operazioni di I/O su disco. Vede il file come un archivio di dati acceduto in modo veloce e memorizzato su dispositivo di I/O in modo efficiente.

La **gestione dei file**, da parte del SO, avviene proprio utilizzando queste due componenti, e ciò per separare il livello file dalle problematiche relative alla memorizzazione e all'accesso efficiente ai dati.

I File System sono organizzati come una *gerarchia* che contiene:

- *struttura delle directory*: contiene informazioni sulle directory associate ai vari utenti;
- *file beta*: i dati sono organizzati secondo una struttura di *record*;
- *file phi*: i dati sono organizzati come un'unica sequenza di byte;

C'è poi la componente che si occupa di effettuare le operazioni di I/O su disco, ovvero l'**IOCS**.



Un file system consiste di due tipi di dati: i dati contenuti nei file (*dati*) e i dati usati per accedere ai file (*dati di controllo o metadati*).

Caratteristiche FS e IOCS

File system

- Strutture delle directory per il raggruppamento conveniente dei file
- Protezione dei file contro gli accessi illeciti
- Semantica condivisione dei file per gli accessi concorrenti a un file
- Memorizzazione affidabile dei file

Sistema di controllo input-output (IOCS)

- Funzionamento efficiente dei dispositivi di I/O
 - Accesso efficiente ai dati in un file
-

FILE

Da un punto di vista di linguaggio di programmazione, un **file** è un oggetto con attributi che descrive l'organizzazione dei suoi dati e i metodi per accedere ai dati.

I **tipi** di file possono essere raggruppati in due *classi*:

- **file strutturati**: collezione di record, ove ogni record è una collezione di campi, e dove ogni campo contiene un singolo dato.
Ogni record si assume contenga un *campo chiave* unico.
- **stream di byte**: sequenza di byte “piatta”.

Un file ha attributi memorizzati nella sua entrata nella directory.

Ad ogni entrata della directory corrisponde un file, e ogni entrata contiene anche delle informazioni sugli attributi di tale file.

Il File System usa le informazioni sugli attributi dei file per localizzarli e per assicurare che ogni operazione svolta su quel file sia consistente con gli attributi del File System.

Operazioni su file

Operazione	Descrizione
Apertura di un file	Il file system recupera l'elemento della directory corrispondente al file e controlla se l'utente il cui processo sta cercando di aprire il file ha i privilegi di accesso necessari per il file. Successivamente, esegue alcune azioni di gestione per avviare l'elaborazione del file.
Leggere o scrivere un record	Il file system considera l'organizzazione del file (Paragrafo 13.3) e implementa le operazioni di lettura/scrittura in maniera appropriata.
Chiusura di un file	L'informazione relativa alla dimensione del file nell'elemento della directory relativo al file viene aggiornata.
Copia di un file	Viene eseguita una copia del file, viene creato un nuovo elemento della directory per la copia e il suo nome, la sua dimensione, la posizione e le informazioni di protezione vengono memorizzate nella voce corrispondente.
Cancellazione del file	L'elemento della directory relativo al file viene cancellato e l'area sul disco occupata viene liberata.
Ridenominazione del file	Il nuovo nome viene registrato nell'elemento della directory relativo al file.
Specificare i privilegi di accesso	Le informazioni di protezione contenute nell'elemento della directory relativo al file vengono aggiornate.

Organizzazione dei file e Metodi di acceso

Usiamo il termine “**pattern di accesso al record**” per descrivere l’ordine in cui un processo accede ai record in un file. Distinguiamo **due modelli** di accesso ai record:

- *sequenziale*: l’accesso ai record avviene nell’ordine in cui si trovano i file;
- *casuale*: l’accesso ai record può avvenire in qualsiasi ordine.

L’**organizzazione dei file** è una combinazione di **due caratteristiche**, cioè il *metodo* con cui si dispongono i record in un file, e la *procedura* per accedere a tali record.

Le caratteristiche di una periferica di I/O determinano l’efficacia per uno specifico pattern di accesso: ad esempio, un nastro è adatto per un accesso sequenziale mentre un disco implementa in modo efficiente sia l’accesso sequenziale che casuale.

Gli accessi ai file sono regolati da una specifica organizzazioni e implementati da un modulo IOCS chiamato *modulo di accesso*.

Un FS supporta diverse organizzazioni, di cui le tre fondamentali sono:

- organizzazione sequenziale;
- organizzazione diretta;
- organizzazione indicizzata.

Organizzazione sequenziale

I record sono memorizzati in sequenza ascendente o discendente sulla base del campo chiave.

Distinguiamo due tipi di *operazioni*:

- lettura del prossimo (o precedente) record;
- saltare il prossimo (o precedente) record.

Tale tipo di organizzazioni è *utile* quando i dati possono essere preordinati in modo conveniente in modo ascendente o discendente.

Inoltre è adatta per i file stream di byte, in quanto accediamo ad ogni byte proprio nell’ordine in cui sono stati scritti.

Organizzazione diretta

Fornisce convenienza/efficienza di elaborazione quando i record sono acceduti in ordine casuale. I file sono chiamati **file ad accesso diretto**.

L'idea è che quando si vuole eseguire un'operazione di lettura o scrittura si deve specificare il *campo chiave*. Tale campo verrà utilizzato per generare l'indirizzo di un record sul dispositivo fisico.

Tale organizzazione ha anche diversi *svantaggi*:

- il calcolo dell'indirizzo dei record consuma tempo di CPU;
- spreco di spazio su disco (su ogni traccia del disco viene memorizzata la stessa quantità di record, tuttavia le tracce esterne sono più grandi e quindi potrebbero contenere più spazio);
- presenza di *record fittizi* per valori chiave che non si usano.

Organizzazione indicizzata

Per l'accesso ai record di un file viene utilizzato un indice, il quale aiuta a determinare l'indirizzo fisico di un record dal valore della chiave.

Possiamo distinguere due ulteriori **tipi** di organizzazione:

■ indicizzata pura

l'indice è una coppia (chiave, indirizzo fisico).

La ricerca non viene effettuata sul file ma sull'indice, e si va ad individuare la chiave ricercata con il suo indirizzo fisico.

Il vantaggio è che se un indice è inferiore al file, questa organizzazione fornisce un'elevata efficienza di accesso poiché una ricerca per indice è più efficiente rispetto a una ricerca nel file.

■ indicizzata sequenziale

È un'organizzazione ibrida che combina gli elementi delle organizzazione indicizzata e sequenziale.

Usa un indice per identificare la sezione del disco che può contenere il record, mentre i record della sezione sono ricercati in modo sequenziali.

Il *vantaggio* è che le operazioni di ricerca vengono effettuate su degli indici che sono comunque piccoli rispetto alla dimensione del file, così da aver un buon compromesso di efficienza di accesso rispetto ad una organizzazione puramente sequenziale.

Inoltre, è migliore rispetto all'indicizzazione pura nel caso la dimensione dell'indice dovesse essere troppo grande.

esempio fra sequenziale e diretto:

- Gli impiegati con numeri 3, 5-9, 11 hanno lasciato l'azienda
- I file ad accesso diretto usano record fittizi per tali record

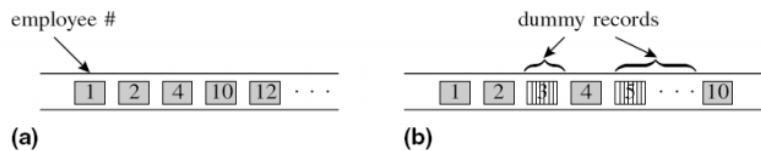
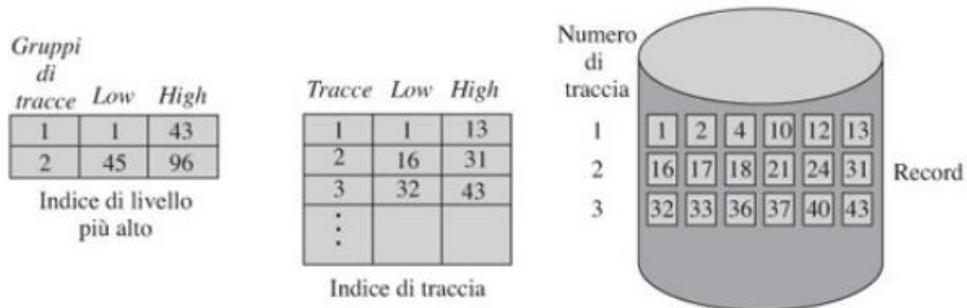


Figure 13.4 Records in (a) sequential file; (b) direct-access file.

esempio indicizzata:



nb: se volessi trovare l'elemento 12, so che dovrei cercare nella traccia 1 perché questa va da 1 a 13. Da qui mi basta fare una ricerca sequenziale per verificare se tale elemento è presente.

Metodi di Accesso

Un *metodo di accesso* è un modulo del IOCS che implementa gli accessi ad una classe di file usando una specifica organizzazione del file. La procedura da utilizzare è determinata dall'organizzazione dei file.

Sono usate **tecniche** avanzate di I/O per l'efficienza:

■ Buffering dei record

I record di un file di input sono letti prima del momento in cui sono necessari ad un processo; così da limitare l'attesa da parte del processo purché avvenga l'operazione di lettura/scrittura.

■ Blocchi di dati

Viene letto o scritto sul dispositivo di I/O un grande blocco di dati, di dimensione maggiore di un record nel file; così da ridurre il numero di operazione di I/O.

DIRECTORY

Le directory sono fondamentalmente delle tabelle, dove ad ogni entrata ci sono informazioni su ciascun file.

Un elemento di una directory è composto da vari **campi**:

Nome del file	Tipo e dimensione	Informazioni sulla posizione	Informazioni sulla protezione	Open count	Lock	Flag	Misc info

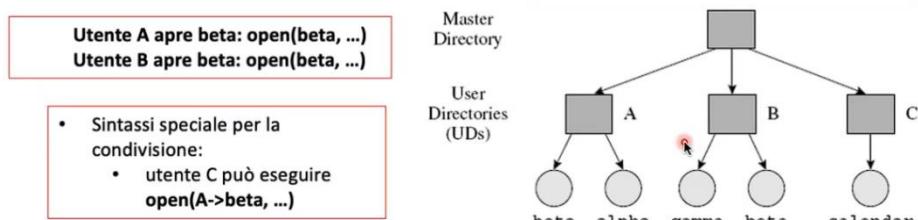
Campo	Descrizione
Nome del file	Nome del file. Se questo campo ha dimensione fissa, i nomi lunghi oltre una certa lunghezza saranno troncati.
Tipo e dimensione	Il tipo e la dimensione del file. In molti file system, il tipo di file è implicito nella sua estensione; per esempio, un file con estensione .c è un file che contiene un programma C e un file con estensione .obj è un file oggetto, che spesso è un file strutturato.
Informazioni sulla posizione	Informazioni sulla posizione del file sul disco. Queste informazioni sono tipicamente sotto forma di una tabella o di una lista concatenata contenente gli indirizzi dei blocchi sul disco allocati al file.
Informazioni sulla protezione	Le informazioni relative agli utenti cui è concesso l'accesso a questo file e in che modo.
Open count	Numero di processi che attualmente accedono al file.
Lock	Indica se un processo sta accedendo al file in maniera esclusiva.
Flag	Informazioni sulla natura del file, quali se il file è una directory, un link o un file system montato.
Misc info	Informazioni varie come l'id del proprietario, la data e l'ora della creazione, l'ultimo accesso e l'ultima modifica.

Un File System ospita file di diversi utenti. Per ogni utente deve concedere: libertà di assegnare i nomi ai file, e condivisione dei file.

Per consentire questo tipo di operazioni, il File System è organizzato in un albero di directory. La prima directory è detta *Master Directory*, e contiene tutte le informazioni associate alle directory dei vari utenti, dette *User Directories*.

Tale suddivisione permette ai vari utenti di poter *rinominare* liberamente i file senza preoccuparsi che il nome possa essere utilizzato da qualche altro utente, perché infatti il percorso per raggiungere quel file è diverso.

Inoltre, un utente può accedere al file di un altro utente con una sintassi speciale (*condivisione*).



ALBERI DELLE DIRECTORY

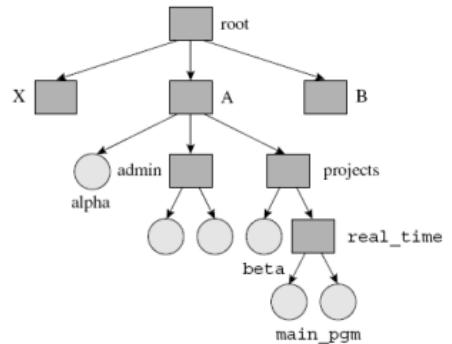
Molti SO moderni sono basati sul struttura ad albero del sistema MULTICS.

Il File System fornisce una directory chiamata root che contiene la **home directory** per *ogni* utente, ovvero una directory che solitamente ha lo stesso nome del nome utente. Un utente organizza le proprie informazioni creando file directory e file dati.

La specifica directory su cui si trova un utente in un dato momento è detta **directory corrente**.

Il File System identifica i file in modo univoco attraverso dei percorsi; tali percorsi sono specificati attraverso dei *pathname*, che in generali si distinguono in:

- *relativi*: eseguiti dalla directory corrente;
- *assoluti*: eseguite dalla directory root.



GRAFI DELLE DIRECTORY

La struttura ad albero porta ad una asimmetria nel modo in cui utenti diversi possono accedere ai file condivisi, in quanto la totale separazione dei file di utenti differenti rende la loro condivisione piuttosto ingombrante.

Per tal motivo si utilizza una *struttura a grafo aciclico*. In questa struttura, un file condiviso può essere puntato dalle directory di tutti gli utenti che hanno accesso ad esso.

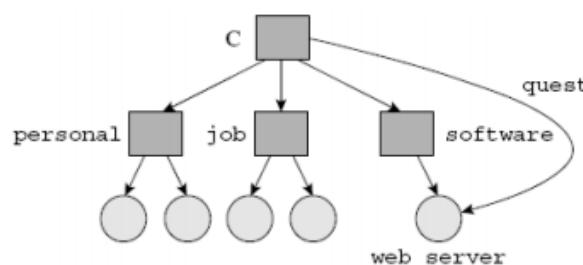
Tali strutture sono implementati attraverso *link*.

Un **link** (collegamento) è una connessione orientata tra due file esistenti nella struttura della directory.

Forma generale: (<from_file_name>,<to_file_name>,<link_name>)

(è tipo un alias che usiamo per abbreviare il nome del percorso)

(~C, ~C/software/web_server, quest)



PROTEZIONE DEI FILE

Gli utenti hanno bisogno di *una condivisione controllata dei file*. Per implementarla, il proprietario di un file specifica quali utenti possono accedere al file e in quale maniera.

Il File System memorizza questa informazione nel campo *protection info* dell'elemento della directory relativo al file.

Soltamente, *protection info* è memorizzato nella lista di controllo accessi, dove ogni elemento della lista è una coppia (<user_name>,<access_privileges>).

Possono essere usati dei gruppi utente per ridurre la dimensione della lista.

I privilegi sono generalmente di tre tipi: *read*, *write* ed *execute*.

ALLOCAZIONE DI SPAZIO SU DISCO

In un disco possono risiedere più FS. Ogni FS è creato su un **disco logico**, ovvero su una *partizione* di un disco.

All'IOCS non può essere nota un'allocazione su disco logico, per tal motivo l'allocazione di spazio sul disco è eseguita dal File System.

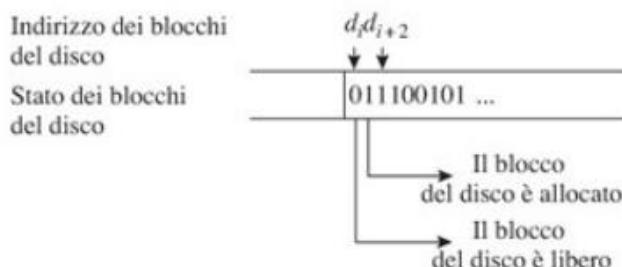
Un tempo il modello di allocazione di memoria era contigua, tuttavia aveva problemi di frammentazione (esterna e interna) ed era complesso da gestire (l'IOCS doveva utilizzare una tabella di blocchi sostitutivi nel caso incontrasse un blocco danneggiato).

Ora il modello di allocazione di memoria è non contigua. Tuttavia il suo utilizzo comporta la gestione di 3 tre problemi:

- **gestire lo spazio libero su disco**: si usa una *free list* o una *disk status map* (DSM);
- **evitare troppi movimenti della testina del disco**: si usano delle *estensioni* (blocchi di disco consecutivi, detti anche cluster) o *gruppi di cilindri*.
- **accedere ai dati nel file**: bisogna gestire le informazioni sullo spazio allocato per la gestione ai dati. Il tipo di accesso dipende dal tipo di approccio: *concatenato* o *indicizzato*.

La **DSM** è una tabella che riferisce quali blocchi del disco sono liberi, utilizzando un bit per ogni blocco. Per tal motivo la DSM è anche detta bit map.

La DSM è consultata ogni volta che deve essere allocato ad un file un nuovo blocco del disco.



ALLOCAZIONE CONCATENATA

Un file è rappresentato da una **lista concatenata di blocchi** del disco. Ogni blocco del disco ha due campi al suo interno: *Dati* e *Metadati*. Il campo Dati contiene i dati scritti nel file, mentre il campo Metadati è il campo di tipo link, che contiene l'indirizzo del prossimo blocco del disco allocato al file.

Lo spazio libero sul disco è rappresentato da una **free list** in cui ogni blocco libero contiene un puntatore al successivo blocco libero. Quando un blocco è richiesto per memorizzare nuovi dati aggiunti al file, un blocco viene estratto dalla free list e aggiunto alla lista dei blocchi del file. Per cancellare un file, la lista di blocchi del file viene semplicemente aggiunta alla free list.

L'allocazione concatenata è **semplice da implementare** e causa di un basso overhead di allocazione/deallocazione. Inoltre, supporta i file sequenziali in modo abbastanza efficiente. Tuttavia, l'accesso ai file con organizzazione non sequenziale non è efficiente.

Anche l'affidabilità è scarsa poiché il danneggiamento del campo dei metadati in un blocco del disco può comportare la perdita dei dati dell'intero file. In modo analogo, il funzionamento del file system può essere influenzato se un puntatore nella free list è danneggiato.

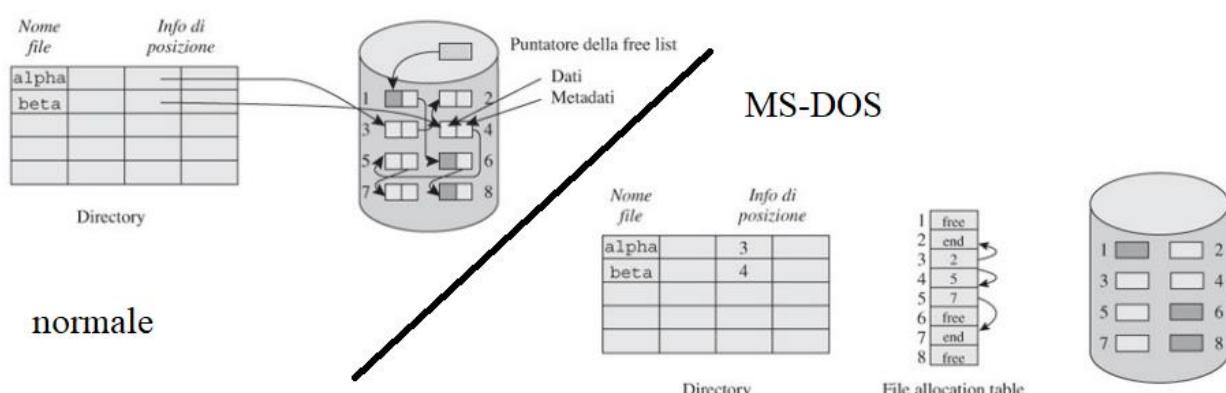
MS-DOS usa una variante di allocazione concatenata che memorizza i metadati separatamente dai dati nel file, e che prende il nome di **FAT**.

FAT ha un elemento corrispondente per ogni blocco del disco.

Lo svantaggio è la necessità di accedere alla FAT per ottenere l'indirizzo del prossimo blocco su disco, soprattutto se consideriamo che la FAT stessa risiede su disco (effettuiamo un'operazione del tipo disco → FAT → disco), il che aggiunge molto overhead.

Una soluzione che velocizza i tempi consiste nel caricare la FAT in memoria (effettuiamo un'operazione del tipo memoria → FAT → disco).

Nonostante l'affidabilità sia migliore, possono presentarsi comunque problemi se la FAT si danneggia prima di essere caricata in memoria.



ALLOCAZIONE INDICIZZATA

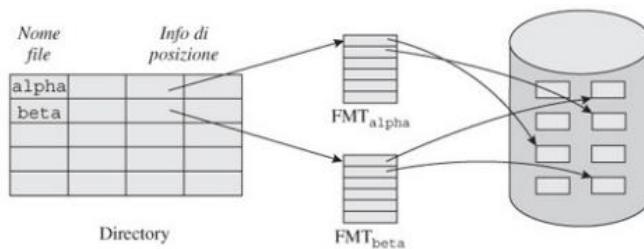
Nell'allocazione indicizzata, un indice chiamato **file map table** (FMT) viene mantenuto per annotare gli indirizzi dei blocchi del disco allocati a un file.

Nella sua forma più semplice, un FMT può essere un array contenente gli indirizzi dei blocchi del disco.

La verifica dei disponibilità dei blocchi può essere fatta facilmente con una DSM.

L'affidabilità è maggiore rispetto all'allocazione concatenata in quanto il danneggiamento di un blocco condanna solo quell'unico blocco.

Rispetto all'allocazione concatenata, l'accesso a un file ad accesso *sequenziale* è meno efficiente poiché bisogna accedere alla FMT di un file per ottenere l'indirizzo del blocco sul disco successivo. Tuttavia, l'accesso ai record in un file ad accesso *diretto* è più efficiente poiché l'indirizzo del blocco che contiene un record specifico può essere ottenuto direttamente dalla FMT.



Inoltre, se i file sono piccoli possono contenere l'intera tabella all'interno di un'entrata della directory corrispondente al file. Invece, se i file sono troppo grandi, non posso fare quest'operazione. Per tal motivo sono nate delle **varianti**.

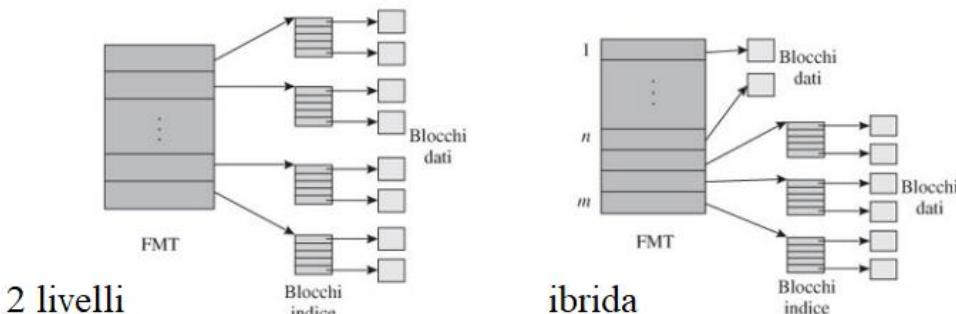
■ Organizzazione FMT a due livelli

La tabella FMT punta ad altri *blocchi indice*, da cui poi ricavo gli indirizzi dei blocchi su disco. È compatta, ma l'accesso ai blocchi di dati è più lento.

■ Organizzazione FMT ibrida

Una parte della tabella FMT indica direttamente i blocchi, mentre un'altra parte utilizza un'organizzazione a due livelli.

Il vantaggio è che se ho piccoli file uso solo la parte ad accesso diretto, altrimenti uso i *blocchi indice*.



INTERFACCIA TRA FS E IOCS

Il file system utilizza IOCS per eseguire le operazioni di I/O.

L'interfaccia tra il file system e IOCS consiste di tre strutture dati: la *File Map Table* (FMT), il *File Control Block* (FCB) e la *Active File Table* (AFT).

In particolare il FCB contiene informazioni per quanto riguarda l'organizzazione del file, le informazioni sulla directory e lo stato corrente dell'elaborazione.

La AFT mantiene gli FCB di tutti i file aperti.

Le operazioni di read/write vengono basate proprio su queste tabelle.

AFFIDABILITÀ DEI FILE SYSTEM

L'affidabilità del file system è il grado di funzionamento corretto di un file system anche nel caso si verifichi un guasto (come corruzione di dati o crash del sistema).

I due **aspetti** principali dell'affidabilità di un FS sono:

- assicurare la correttezza della creazione, cancellazione e aggiornamenti dei file;
- prevenire la perdita di dati nel file.

Notare che: un *guasto* è un difetto in una qualche parte del sistema, e il suo verificarsi causa un malfunzionamento; Un *malfunzionamento* è un comportamento erroneo del sistema.

PERDITA DI CONSISTENZA DEL FILE SYSTEM

La **consistenza** del FS implica la correttezza dei metadati e delle operazioni del file system.

Una perdita di consistenza vorrebbe dire che la correttezza dei metadati e delle operazioni viene persa, e ciò non ci consente di reperire i dati o di non reperire quelli corretti.

Un guasto può causare i seguenti malfunzionamenti:

- qualche dato di un file aperto può essere perso;
- parte di un file aperto può diventare inaccessibile;
- i contenuti di due file possono essere scambiati.

(*esempio 1*) si supponga di aggiungere un blocco del disco ad un file. Se supponiamo di inserire tale blocco d_j fra due blocchi d_1 e d_2 dove d_1 punta a d_2 : normalmente dovremmo avere una situazione del tipo $d_1 \rightarrow d_j \rightarrow d_2$; tuttavia se avviene un guasto potrebbe capitare una situazione del tipo $d_1 \rightarrow d_j \rightarrow \text{niente}$; $\text{niente} \rightarrow d_2$, ovvero d_2 non è più accessibile.

(*esempio 2*) I contenuti di due file possono essere scambiati se i metadati sono salvati solo dopo una close del file.

APPROCCI PER L'AFFIDABILITÀ DEL FS

Vengono utilizzati due approcci per garantire l'affidabilità:

Approccio	Descrizione
Recupero	Ripristinare i dati e i metadati del file system a un precedente stato consistente.
Fault tolerance	Prevenzione contro la perdita di consistenza dei dati e dei metadati in seguito a guasti, in modo che il funzionamento del sistema sia corretto in ogni situazione, ovvero, non si verificano perdite o corruzioni di dati.

Il *recupero* è un approccio classico che è attivato quando si osserva un malfunzionamento. La *tolleranza ai guasti* fornisce sempre operazioni corrette nel FS.

LEZIONE 24 – IOCS: INPUT OUTPUT CONTROL SYSTEM

L'IOCS implementa le operazioni sui file. Ha due obiettivi principali:

- l'implementazione efficiente delle attività di elaborazione di un file in un processo;
- throughput elevato dei dispositivi di I/O.

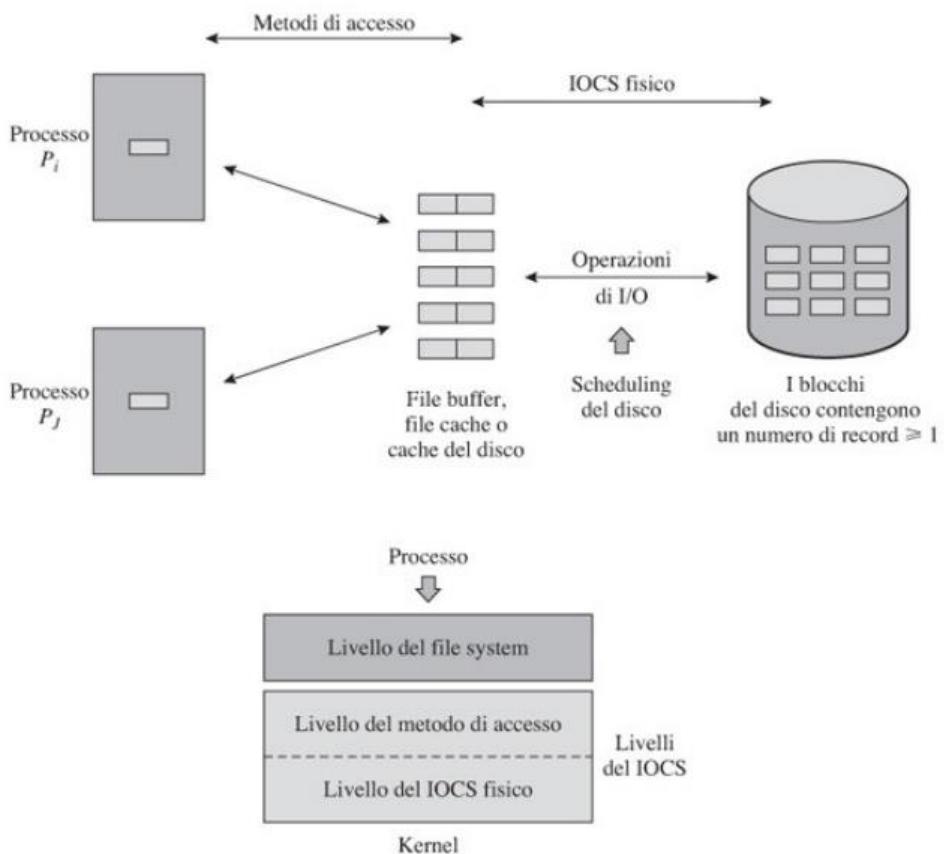
Per realizzare questi due obiettivi, l'IOCS è organizzato in due livelli chiamati *metodi di accesso* e *IOCS fisico*.

I *metodi di accesso* gestiscono le operazioni di lettura/scrittura dei dati, mentre l'implementazione fisica di tali operazioni viene effettuata dall'IOCS fisico.

L'IOCS fisico utilizza politiche di scheduling per migliorare il throughput dei dispositivi di I/O coinvolti.

Questa separazione a due livelli permette di separare le problematiche relative all'implementazione di file a livello di processo da quelli a livello di dispositivo.

L'IOCS fisico nei SO più vecchi si trovava all'interno del kernel, mentre in quelli moderni si trova al di fuori per un motivo di estendibilità. In quest'ultimo caso deve essere richiamato attraverso delle System Call.



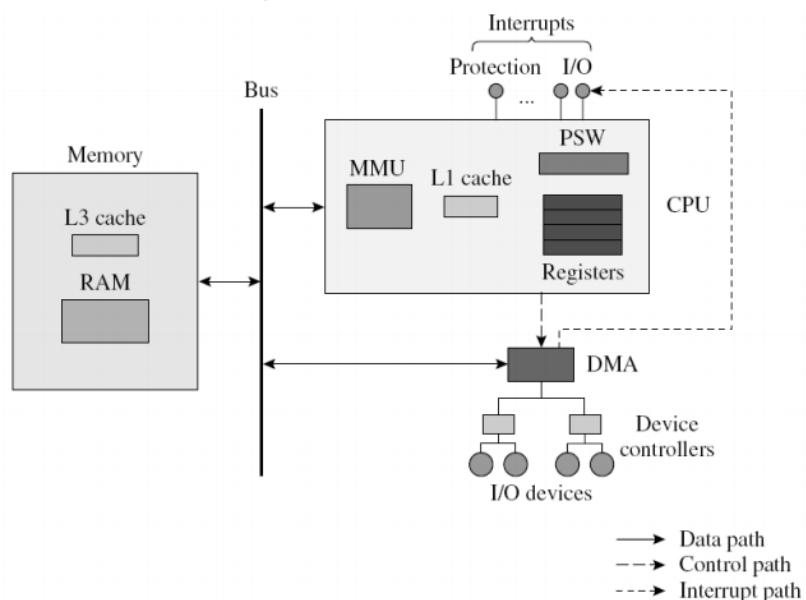
ORGANIZZAZIONE DELL'I/O

Il sottosistema di I/O ha un percorso dai dati in memoria indipendente.

I dispositivi sono connessi ai controller dei dispositivi che sono connessi al DMA. Un dispositivo è identificativo dalla coppia (*controller id, device id*), ovvero l'id del controller e l'id di quella classe del controller.

La CPU indica che è necessario effettuare un'operazione di I/O. Da tale istruzione si punta un'area di memoria dove sono presenti una serie di comandi necessari per la realizzazione di tale operazione.

Un'operazione di I/O viene implementata dal DMA, un controller di dispositivo e dal dispositivo stesso (ma non dalla CPU, che può invece effettuare altre operazioni).



Operazioni di I/O

I comandi di I/O sono tipicamente memorizzati in memoria, e l'indirizzo dell'area di memoria che contiene i comandi viene utilizzato come operando dell'istruzione di I/O che la CPU invia al DMA.

Per esempio, se si vuole effettuare una *read* su un blocco del disco (*track_id, block_id*), si dovrà effettuare un'operazione tipo:

I/O-init(controller_id, controller_device), I/O_command_addr

dove *I/O_command_addr* è l'indirizzo di partenza dell'area di memoria che contiene due comandi:

1. posiziona la testina del disco sulla traccia *track_id*;
2. legge lo specifico record.

Third Party DMA (caso sopra)

In una *DMA di terze parti*, quando è eseguita un'istruzione di I/O, succede che il controller del DMA passa i dettagli dei comandi di I/O al controller del dispositivo di I/O. Il dispositivo quindi consegna i dati al controller di dispositivo.

Il trasferimento dei dati da controller del dispositivo in memoria avviene in diversi passi:

1. Il controller del dispositivo invia un segnale DMA, detto DMA request, quando è pronto al trasferimento.
2. il DMA, ricevuto il segnale, ottiene il controllo del bus e vi pone l'indirizzo di memoria che partecipa al trasferimento. Infine, invia un DMA ack al controller del dispositivo.
3. Il controller del dispositivo trasferisce i dati verso o dalla memoria.
4. Alla fine del trasferimento, il controller del DMA genera un interrupt di completamente I/O con codice uguale all'indirizzo del dispositivo.

La routine di servizio degli interrupt analizza il codice per trovare qualche dispositivo che ha completato la sua operazione di I/O e intraprende le operazioni appropriate.

In tutto questo possiamo notare come la CPU non intervenga, però questo potrebbe causare dei problemi. Infatti il bus fra CPU, memoria e DMA è condiviso, quindi il bus potrebbe essere *conteso*.

Per evitare competizioni che potrebbero causare eccessivi ritardi, si adotta una *tecnica speciale* che permette sia alla CPU che al DMA di usare il bus senza creare grossi ritardi. Fondamentalmente la CPU usa dei punti speciali per favorire il DMA, e il DMA attende il verificarsi di uno di questi punti per usare il bus.

DISPOSITIVI DI I/O

Esistono differenti tipologie di dispositivi di I/O che funzionano sulla base di differenti principi fisici, come: la *generazione di segnali elettromeccanici* e la *memorizzazione dati ottica o elettromagnetica*.

I dispositivi di I/O possono essere classificati sulla base di 3 criteri:

- **scopo**: dispositivi di input, stampa e di memorizzazione;
- **natura dell'accesso**: sequenziale (tastiera, mouse, rete, nastro) o casuale (dischi);
- **modalità di trasferimento dati**: caratteri o blocchi.

L'informazione letta o scritta in un comando di I/O costituisce un record.

Modalità di trasferimento dati

La modalità di trasferimento dipende dalla velocità di trasferimento stessa.

■ Dispositivi di I/O lento

Lavorano nella modalità carattere: è trasferito un carattere per volta tra memoria e periferica. Sono ad esempio tastiere, mouse e stampanti.

Contiene un buffer che memorizza il carattere.

Il controller genera un interrupt in conseguenza di una lettura dal buffer (dispositivo di input) o di una scrittura nel buffer (dispositivo di output).

I controller possono essere connessi direttamente al bus.

■ Dispositivi di I/O veloci

Lavorano nella modalità blocco. Sono ad esempio nastri e dischi.

Tali dispositivi sono connessi ad un controller di DMA e devono trasferire i dati a specifiche velocità. Per via di tale esigenza di velocità, per evitare problemi, i dati sono trasferiti tra la periferica di I/O e un *buffer* del DMA.

Tempo di accesso e tempo di trasferimento

Il tempo di esecuzione di un dispositivo di I/O può essere ottenuto come la somma di due componenti: il *tempo di accesso* e il *tempo di trasferimento*.

In particolare diremo che:

- **tempo di I/O (t_{io})**: intervallo di tempo tra l'avvio dell'istruzione di I/O e il completamento dell'operazione.
- **tempo di accesso (t_a)**: intervallo di tempo tra un comando read o write e l'inizio del trasferimento.
- **tempo di trasferimento (t_x)**: intervallo di tempo necessario per trasferire dei dati da/verso una periferica durante un'operazione read o write (inizio trasferimento primo byte, fine trasferimento ultimo byte).

Quindi: $t_{io} = t_a + t_x$



nb: Il tempo di accesso è *costante* per i dispositivi sequenziali (ci si sposta a sinistra e a destra), mentre è *variabile* per i dispositivi ad accesso casuale.

Individuazione e correzione errori

Gli errori possono verificarsi durante la scrittura o la lettura dei dati o durante il trasferimento tra un dispositivo di I/O e la memoria.

I dati trasmessi sono visti come un flusso di bit, i quali sono rappresentati attraverso dei *codici speciali*. In questo modo possono essere utilizzate delle tecniche per l'individuazione e correzione degli errori.

Per l'**individuazione degli errori** l'idea è quella di memorizzare informazioni ridondanti con i dati, determinate dai dati con *tecniche standard*, così che quando si leggono delle informazioni da una periferica di I/O sono lette anche le informazioni di individuazione degli errori.

Inoltre, tali informazioni sono calcolate nuovamente dai dati letti, usando la stessa tecnica: si confrontano le info lette dal mezzo di I/O e quelle determinate dai dati letti, ed un eventuale mismatch indica l'occorrenza di un errore di memorizzazione.

La **correzione dell'errore** è fatta in modo analogo: si memorizzano informazioni aggiuntive così che algoritmi appositi possano individuare l'errore e correggerlo.

La memorizzazione e la lettura di informazioni ridondanti tuttavia causa **overhead**, e un eventuale correzione comporterebbe overhead ancora maggiore.

Per l'individuazione e la correzione possono essere usati diversi **approcci**:

- **bit di parità:** quando viene memorizzato un byte di informazione, vi si associa anche un'informazione aggiuntiva (il bit di parità) che indica se il numero di bit del byte è pari oppure dispari.

Così facendo, tutti gli errori su singolo bit possono essere individuati (anche quello sul bit di parità stesso). Ovviamente se c'è errore su più di un bit tale strategia non è più efficace.

- **Controllo di ridondanza ciclico (CRC):** utilizza una funzione hash per rilevare errori su più bit.

In ambo gli approcci si usa l'aritmetica modulo-2, nella quale l'addizione è rappresentata come un OR-esclusivo.

DISCO MAGNETICO

L'elemento di memorizzazione è un oggetto circolare chiamato **piatto** che ruota intorno al proprio asse. La superficie circolare è ricoperta di materiale magnetico. Tutti i piatti ruotano grazie ad un *motore*.

Ogni **testina** è sospesa sopra un piatto, sono montate su un braccio detto *attuatore* e si muovono in blocco.

Per ottimizzare l'uso della superficie del disco, i piatti sono divisi in anelli detti tracce, le quali a loro volta sono organizzate in **settori** di dimensione standard.

La loro suddivisione può essere parte dello HW (hard sectoring) o implementata da software (soft sectoring).

È marcata la posizione di avvio su ogni traccia ed ai record di una traccia sono attribuiti numeri seriali rispetto a tale posizione.

Ciò significa che il disco può accedere ad ogni record con un indirizzo del tipo (*numero traccia, numero record*).

Una singola testina di lettura/scrittura registra e legge dalle *tracce* della superficie. L'insieme delle tracce alla stessa distanza dal centro di ogni piatto costituiscono un **cilindro**.

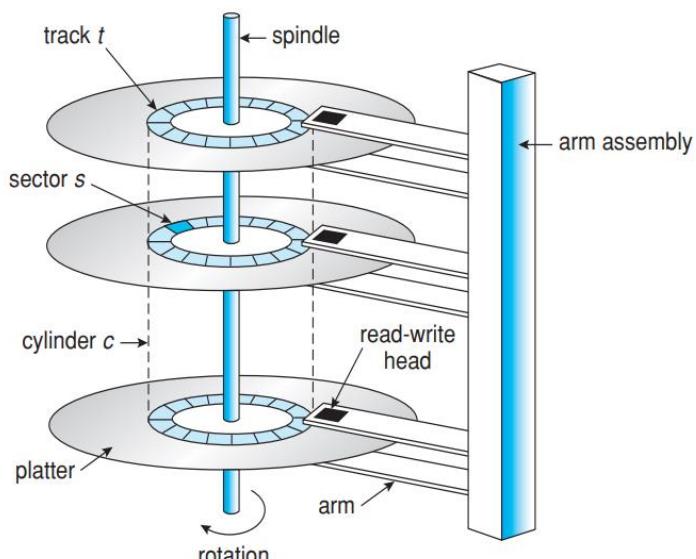
Siccome le testine sono fisse e si muovono in blocco, tutte le tracce di un **cilindro** possono essere accedute dalla stessa posizione della testina. Quindi l'uso della nozione di cilindro consente di ridurre il numero di movimenti della testina.

Infatti, se devo memorizzare dei dati adiacenti di un file mi conviene memorizzarli sullo stesso cilindro.

Ciò significa che possiamo vedere l'indirizzo di un record come: (*numero cilindro, numero superficie, numero record*).

I byte vengono memorizzati in modo seriale lungo una traccia circolare sulla superficie del disco, quindi la testina si muove radialmente lungo il piatto.

Per il rilevamento degli errori, in un disco, viene utilizzato il CRC.



Il *tempo di accesso* è $t_a = t_s + t_r$, dove:

- t_s tempo di ricerca: tempo per posizionare la testina sulla traccia richiesta;
- t_r latenza rotazionale: tempo per accedere al record desiderato sulla traccia.

La latenza rotazionale media è il tempo richiesto per una rivoluzione di metà del disco (in media 3-4 ms).

È possibile aumentare la capacità dei dischi aumentando il numero di piatti, tuttavia maggiore è il numero di dischi, maggiore sarà il numero di testine e di conseguenza maggiore sarà il peso dell'attuatore.

Più pesante è l'attuatore, maggiore sarà lo stress a cui sarà sottoposto il disco magnetico (per tal motivo, non si va oltre un certo numero di piatti).

ORGANIZZAZIONE DEI DATI SU DISCO

I dati devono essere organizzati in modo da garantire un accesso efficiente.

Infatti il tempo di trasferimento, di commutazione e di ricerca possono crescere se non si adottano misure preventive.

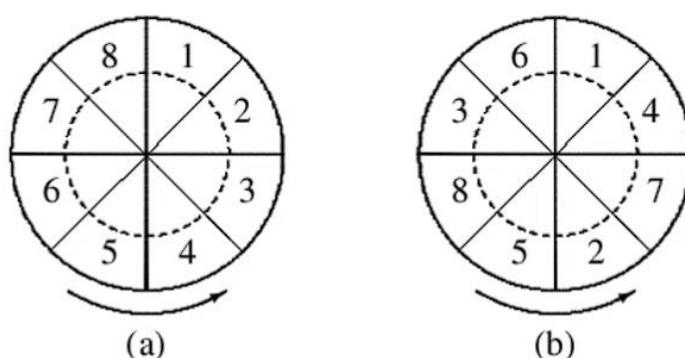
Il problema generale che si verifica è che il disco continua a girare per tutto il tempo, e quello che vorremmo evitare è che se per accedere ad un dato adiacente nel disco dobbiamo aspettare il completamento di tutta la rivoluzione del disco stesso.

Ciò provoca dei ritardi che possono evitare usando alcune **tecniche**: l'*alternanza dei settori*, la *testina asimmetrica* o il *cilindro asimmetrico*.

Alternanza dei settori

In tale tecnica un piccolo numero di settori sono saltati mentre si memorizzano record adiacenti di un file.

Il numero di settori saltati è chiamato fattore di alternanza (inf).



(a) nessuna alternanza; i record adiacenti in un file occupano settori adiacenti

(b) fattore di alternanza = 2; ci sono due settori tra record adiacenti

Testina asimmetrica

Il disco richiede del tempo per commutare dalla lettura dei dati di una traccia ai dati di un'altra traccia in un cilindro (tempo commutazione testina), ed alcuni settori passano sotto la testina durante questo tempo.

Tale tecnica organizza le posizioni di inizio traccia su piatti diversi di un cilindro in modo tale che questo sfasamento possa contrattare il movimento della testina così da non perdere il settore considerato.

Cilindro asimmetrico

Il disco ruota mentre le testine si spostano sulle tracce di un cilindro adiacente. Per tal motivo si organizzano le posizioni di inizio traccia su cilindri consecutivi così da tener conto del tempo di ricerca dopo la lettura dell'ultimo settore su un cilindro.

DISCHI RAID (Redundant Array of Independent Disks)

La RAID è una tecnologia che consiste nell'utilizzare un array di dischi poco costosi anziché un unico disco.

Sono usate diverse disposizioni per fornire **tre benefici**:

- **affidabilità**: memorizza i dati in modo ridondante. Ciò consente anche una maggiore velocità in quanto i record di dati ridondanti sono letti/scritti in parallelo;
- **tassi veloci di trasferimento dati**: memorizza i dati dei file su più dischi nel RAID. Ciò comporta una maggior velocità di *trasferimento* in quanto i dati sono letti/scritti in parallelo.
- **accesso veloce**: memorizza due o più copie dei dati. Per leggere i dati, accede alla copia che è accessibile in modo più efficiente.

La memorizzazione in un raid viene eseguita usando una *disk stripe*.

Una **disk stripe** è una collezione di *strip* posizionate allo stesso modo su dischi diversi nel RAID, e dove una *disk strip* è un'unità di dati su disco, come un settore, un blocco o una traccia.

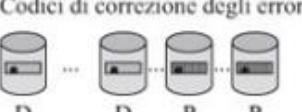
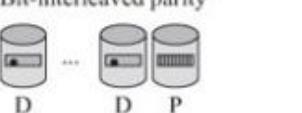
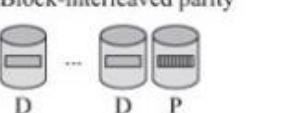
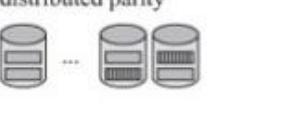
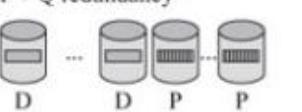
I dati scritti sugli *strip* in uno *stripe* possono essere letti in parallelo. Questa disposizione fornisce elevati tassi di trasferimento.

Livelli RAID

I livelli RAID sono configurazioni RAID che utilizzano differenti tecniche di ridondanza e organizzazione di disk striping:

- **RAID 0** (striping del disco): le strip che costituiscono una stripe sono tutte scritte su dischi diverse. Quindi quando effettuo un'operazione di trasferimento file, leggo ciascuna strip di una stripe in parallelo.
Si hanno tassi di trasferimento elevati, ma non si pone alcuna attenzione sull'affidabilità.
- **RAID 1** (mirroring del disco): Gli stessi dati sono scritti su due dischi, e qualora si volesse leggere un'informazione viene prelevata la copia accessibile in modo più veloce.
Si hanno tassi di trasferimenti meno veloci rispetto al RAID 0, ma si ha maggiore affidabilità.
- **RAID 0+1**: avviene prima lo striping del disco, e poi il mirroring;
- **RAID 1+0**: avviene prima il mirroring, e poi lo striping del disco.
Fornisce una miglior affidabilità del RAID 0+1.

Si hanno inoltre **ulteriori livelli**, i quali utilizzano più precauzioni per garantire l'affidabilità:

Livello 2	Codici di correzione degli errori 	Le informazioni di ridondanza sono memorizzate per rilevare e correggere gli errori. Ogni bit di dati o di informazione ridondante è memorizzato su un disco differente e letto o scritto in parallelo. Garantisce tassi di trasferimento elevati.
Livello 3	Bit-interleaved parity 	Analogo al livello 2, fatta eccezione per il fatto che utilizza un singolo disco di parità per la correzione degli errori. Un errore che si verifica durante la lettura dei dati da un disco viene rilevato dal controller. Il bit di parità viene utilizzato per ripristinare i dati persi.
Livello 4	Block-interleaved parity 	Scrive un <i>blocco</i> di dati, ovvero byte di dati consecutivi, in una strip e calcola una singola strip di parità per le strip di una stripe. Garantisce tassi di trasferimento elevati per operazioni di lettura molto grandi. Le operazioni di lettura piccole hanno bassi tassi di trasferimento; tuttavia, molte di queste operazioni possono essere eseguite in parallelo.
Livello 5	Block-interleaved distributed parity 	Analogo al livello 4, fatta eccezione per il fatto che le informazioni sono distribuite su tutti i dischi. Consente di evitare che il disco di parità diventi un collo di bottiglia per l'I/O come nel livello 4. Inoltre garantisce migliori prestazioni in lettura rispetto al livello 4.
Livello 6	P + Q redundancy 	Analogo al livello 5, fatta eccezione per il fatto che utilizza due schemi di parità distribuita indipendenti. Supporta il ripristino dal guasto di due dischi.

DRIVER DI DISPOSITIVO

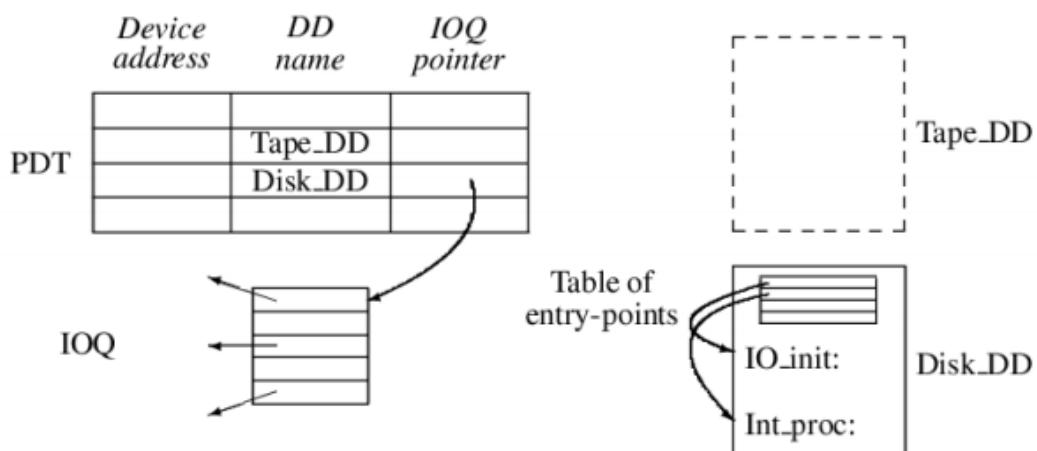
L'IOCS fisico potrebbe far parte del kernel, e l'aggiunta di una classe può essere complicato in quanto è come se dovessi modificare parte del kernel. Per tal motivo i moderni SO fanno uso dei *driver di dispositivo*, i quali possono essere facilmente aggiunti. I driver di dispositivo sono caricati in memoria in fase di *boot* o all'occorrenza del SO.

Purché l'IOCS fisico possa basarsi sui driver di dispositivo ha bisogno di una certa **organizzazione**.

In particolare, è necessario usare delle strutture dati che sono coinvolte tra l'interazione fra i driver e l'IOCS fisico.

Si ha una **tabella di dispositivi fisici PDT**, dove ciascuna entrata della tabella contiene l'indirizzo del driver, il nome del driver (DD name) e un puntatore ad un'altra struttura, detta coda di I/O (IOQ pointer).

Un driver può contenere delle funzioni ad inizio tabella. Quando il IOCS fisico è richiamato per iniziare un'operazione di I/O, localizza l'elemento relativo al dispositivo nella PDT ed esegue la generica operazione di inserimento dei dettagli dell'operazione di I/O nella IOQ del dispositivo. A questo punto consulta il campo nome DD dell'elemento della PDT, ottiene l'identità del driver di dispositivo e lo carica in memoria, se non è già stato caricato.



SCHEDULING DEL DISCO

L'IOCS e i driver di dispositivo utilizzano delle politiche di scheduling, necessarie per ottimizzare il throughput del disco.

Ne distinguiamo diverse:

- **First-come, first-served** (FCFS): seleziona l'operazione di I/O con tempo di richiesta inferiore;
- **Shortest seek time first** (SSTF): seleziona l'operazione di I/O con il più breve tempo di ricerca rispetto alla posizione corrente delle testine del disco;
- **SCAN**: questa politica muove le testine del disco da un estremo all'altro del piatto, servendo le operazioni di I/O per i blocchi su ogni traccia/cilindro prima di spostarsi sulla prossima traccia/cilindro. Per questo motivo si chiama scan. Quando le testine del disco raggiungono l'altra estremità del piatto, la direzione di movimento viene invertita e le nuove richieste vengono servite nella scansione inversa

Una variante chiamata look inverte la direzione delle testine del disco quando non ci sono più richieste di I/O nella direzione corrente;

- **Circular SCAN o CSCAN**: questa politica esegue la scansione come nello scheduling SCAN. Tuttavia, non esegue mai la scansione inversa;
La variante circular look (C-look) muove le testine solo finché ci sono richieste da eseguire prima di iniziare una nuova scansione.

In generale non c'è un metodo migliore, ma dipende molto come si presentano le richieste. Anche se le politiche *look* e *C-look* sono quelle più utilizzate.

Esempio di esercizio

- Supponiamo che nella coda delle richieste di un'unità disco composta da 200 tracce si trovano le richieste di dati nei blocchi
 - 39700 – 304 – 115 – 2600 – 2120 – 270 – 321 – 0 – 760 – 20000
 - il blocco *i-esimo* è memorizzato nella traccia ***i mod 200***
- La testina ha eseguito l'ultimo movimento portandosi dalla traccia 85 alla traccia **97**
- Si ipotizzi che lo spostamento da una traccia ad un'altra richieda tempo medio pari a 40 μ s per traccia, che l'inversione della direzione di movimento richieda in media 80 μ s e la velocità di rotazione sia di 7200 giri
- Si vuole determinare il **tempo richiesto, complessivamente, per accedere alle tracce** indicate per le politiche SSTF, C-SCAN e LOOK.

Soluzione

- Latenza rotazionale: $60/(2 \times 7200) = 4.17 \text{ ms}$
 - Dobbiamo determinare la traccia alla quale si trova il blocco (***i mod 200***)
 - Blocchi: 39700 – 304 – 115 – 2600 – 2120 – 270 – 321 – 0 – 760 – 20000
 - Tracce: 100 – 104 - 115 - 0 - 120 - 70 - 121 - 200 - 160 - 0
1. SSTF. La sequenza di scheduling è:
 - 97 100 104 115 120 121 160 200 70 0
 - Le distanze tra tracce della sequenza sono: 3 4 11 5 1 39 40 130 70
 - Il tempo di accesso è $t_a = (303 \times 40 \mu\text{s}) + 80 \mu\text{s} + (9 \times 4.17 \text{ ms}) = 12.12 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 49.73 \text{ ms}$
 2. C-SCAN
 - 97 100 104 115 120 121 160 200 0 70
 - Le distanze tra le tracce
 - 3 4 11 5 1 39 40 200 70
 - $t_a = (373 \times 40 \mu\text{s}) + 80 \mu\text{s} + (9 \times 4.17 \text{ ms}) = 14.92 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 52.53 \text{ ms}$
 3. LOOK
 - 97 100 104 115 120 121 160 200 70 0
 - 3 4 11 5 1 39 40 130 70
 - $t_a = (303 \times 40 \mu\text{s}) + 80 \mu\text{s} + (9 \times 4.17 \text{ ms}) = 12.12 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 49.73 \text{ ms}$