Sistemi Operativi

venerdì 9 luglio 2021 17:00

Sistemi Operativi

LEZIONE 1-2 (introduzione al corso)

LEZIONE 3 (compiti del sistema operativo)

COMPITI DEL S.O.

Cosa deve fare un sistema operativo?

- Fornire un ambiente per fare eseguire un programma
  N.B. All'interno del S.O. non si parla più di programma(cioè un elenco di istruzioni e dati da elaborare) ma di processo (cioè un programma in esecuzione a cui vengono assegnate delle entità *dinamiche*, gli oggetti del S.O. --> tra cui il processore, la RAM e i dispositivi di I/O)

- Gestione dei processi:
  - Creazione

- Cancellazione
- Sospendere/riattivare
- Collaborazione --> sincronizzazione/comunicazione

Qual è la caratteristica più importante che ci fa capire se si parla di programma o processo? Il Program Counter
N.B. Ogni processo crederà di essere l'unico in esecuzione sul processore (se due processi vogliono comunicare, allora interverrà il S.O.)

- Gestione della memoria principale:
  Quando io ho un nuovo programma che diventerà un processo e che quindi deve essere allocato in memoria, dove lo vado a mettere? Se ci sono altri programmi che girano non ho tutto lo spazio disponibile e quindi ho la necessità di individuare la fetta di memoria adatta allo scopo.
  - Allocazione/deallocazione
  - Spazio libero
  - Richieste processi --> con utilizzo di memoria secondaria (ROM, memoria virtuale) se finisce la memoria principale (RAM)
  - Tecniche di allocazione

- Gestione dei dispositivi di input/output:
  Il sistema operativo deve interfacciarsi all'hardware:
  - Interfaccia HW
  - dispositivo

  Per farlo ha bisogno di una parte software:
  - Driver --> spiega al S.O. cosa si aspetta di ricevere l'HW

  Memoria secondaria (classificata anche questa un dispositivo di I/O):
  Il sistema operativo ha una serie di servizi per la sua gestione:
  - Allocare(mount) / deallocare(dismount)
  - Ottimizzare
  - Spazio libero

  Il sistema operativo dovrà anche gestire i file:
  Operazioni:
  - Creare/cancellare
  - Manipolare
  - Apertura/chiusura

  Tramite una visione logica (indipendente dal supporto), il S.O. vi mostra sempre il concetto di file rappresentato in modo uniforme (che si tratti di cd/SSD/HDD/…)

  A seconda del sistema operativo ci saranno visioni logiche diverse:
  Windows:
  - C:\
  - D:\
  - …
  - A:\ e B:\ in passato venivano utilizzati per i floppy disk, dopodiché si è passati al C:\ per l'HDD

  Unix/Linux:
  - (struttura ad albero, [radice: root = /])

- Deve garantire che tutti siano autorizzati tramite dei meccanismi di protezione:
  - Multi utente
  - Multi tasking (+ processi)

  Meccanismi:
  - Autenticazione: chi entra nel sistema è chi dice di essere
  - Autorizzazioni: danno il controllo delle varie risorse
  - Controllo risorse: devono essere gestite e protette

- Interfaccia utente
  Deve essere facile da usare!
  Modalità:
  - Interprete command line (Amministrazione) --> Shell
  - GUI

LEZIONE 4 (servizi e storia dei sistemi operativi)

SERVIZI NECESSARI PER IL S.O.
Possiamo distinguere diverse categorie di servizi:
- Funzioni visibili all'utente (comodità)
  - Esecuzione di un programma
  - Coordinamento I/O
  - Gestione del file system
  - Comunicazione processi/reti
  - Gestione degli errori (HW, I/O, programmi, …)
- Funzioni non visibili all'utente (efficienza)
  - Allocazione risorse (es: CPU --> scheduling)
  - Contabilità (log): utili per fare delle statistiche
  - Protezione e sicurezza

Chiamate di sistema (SYSTEM CALL)

Sono delle chiamate che servono al programmatore per utilizzare i servizi del sistema
Meccanismi di chiamata di una system call::

- Tramite semplici chiamate alle librerie
- chiamando degli interrupt tramite superutente
- Utilizzare un programma che chiama delle librerie ad alto livello, le quali internamente chiamano le system call di un certo S.O. (basta un unico programma per più S.O.

Esempi:

- Controllo dei processi (crea processo: Fork();)
- Gestione I/O (richiedi/rilascia risorsa, open/close, read/write)
- Gestione file

Comunicazione

- Scambio di messaggi (send/receive)
- Memoria condivisa (tra più processi)
- Semafori (meccanismi per far aspettare fino al momento giusto un processo)

STORIA DEI S.O. (UNIX ~ LINUX)

Aspetti caratterizzanti nell'evoluzione di unix e linux:

- 1965, USA: MIT --> Time Sharing, Calcolo scientifico (sistema multiutente e multitasking) [S.O. scritto in assembly] <---> MULTICS
- 1965: LAB BELL --> hanno preso MULTICS e lo hanno semplificato <---> UNICS
- 1970: UNICS viene scritto in un linguaggio ad alto livello<---> UNIX
  Ideato il linguaggio C (ANSI C di Ritchie)
  Software UNIX gratuito per università
- 1980: AT&T --> System III V
- 1978, West Coast, Berkeley + Darpa: Berkeley software distribution (BSD), insieme al quale nacquero:
  - Gestione della memoria (paginazione, virtuale)
  - Protocolli di rete TCP/IP (comunicazione)
  - CSH (C Shell)
  - Compilatori
- 1980: Lab BELL + BSD ==> standardizzazioni (IEEE --> Posix)
  Grazie alla standardizzazione molte interfacce si sono unificate:
  - AT&T
  - BSD
  - IBM + DEC + HP
- 1987, Tanembaum: MINIX (caratteristica = micro kernel <-- 26k righe di codice)
  - Approccio diverso, tutte le attività meno essenziali erano delegate a dei moduli (svantaggio prestazione, + servizi da attrav ersare = perdita di tempo)
- 1991: LINUX
- 1994: Primo kernel LINUX 1.0

PROGETTAZIONE DI UN SISTEMA OPERATIVO

Struttura tipica di un sistema operativo (CAP 2.8):
Problema: più cresce il sistema più si perde la sua struttura, diventa così ingestibile, così è spesso necessario rifarlo da zero!

Tipi di distema:

- SISTEMA MONOLITICO = senza struttura
  Esempio: Unix originale
  PRO
  Gestione software = ok

  CONTRO
  Efficienza = not ok

- SISTEMA A LIVELLI / STRATI
  PRO
  Moduli stratificati:
  - Modularità --> interfacce
  - Black box
  - Debug / verifica

  CONTRO
  Problema 1 : difficoltà ad identificare i livelli, non è facile trovare una dipendenza di livelli a cascata
  Problema 2: efficienza, ci sono troppe interfacce da attraversare

NOTA (altri sistemi):
La gestione modulare è quella più flessibile
Partendo dalle idee della programmazione ad oggetti costruiamo altri S.O.

Benefici:

- Interfacce
- Creazione oggetti

LEZIONE 5 (virtualizzazione, gestione CPU e processi)

VIRTUALIZZAZIONE (CAP 1.7)
La virtualizzazione si occupa di creare una macchina virtuale e simulare dell'hardware.

Il virtualizzatore è un programma che si occupa di convertire tutte le istruzioni per l'hardware_virtuale in istruzioni equivalenti sull'hardware fisico a disposizione.
Possono coesistere più macchine virtuali simultaneamente, il virtualizzatore si occupa di allocare diverse quantità di memoria, cpu,… per ogni vm

Esempio:
HARDWARE_1 = [HARDWARE_1 VIRTUALE --> VIRTUALIZZATORE --> HARDWARE] --> SOFTWARE
HARDWARE_2 = [HARDWARE_2 VIRTUALE --> VIRTUALIZZATORE --> HARDWARE] --> SOFTWARE

VANTAGGI:
- ottimizzazione di risorse hardware (cpu, memoria, …)
- le vm create sono indipendenti --> protezione di risorse virtuali
- Utile per simulazione / testing (di S.O. per esempio)

Domande:
Qual è l'obiettivo che hanno in comune virtualizzatore e S.O.?
- Ottimizzare l'hardware

In cosa un virtualizzatore è simile ad un S.O.?
Necessità molto simili:
- Scheduling delle varie vm
- Gestione della memoria principale/secondaria, dispositivi I/O

In cosa un virtualizzatore si differenzia da un S.O.?
- La vm si occupa di simulare un ambiente virtuale
- Il S.O. invece offre un ambiente per l'esecuzione dei programmi

JAVA VM
Si limita a fare una simulazione software: è un processo del sistema all'interno del S.O.

AVVIO DEL S.O. (CAP 2.9)
All'avvio dell'hardware la macchina è senza S.O. quindi devo trovarmi un S.O. funzionante! --> fase a basso livello necessaria per il caricamento del sistema

Fasi per permettere l'avvio del sistema:
- Controllo hardware
- Bootstrap: carica il sistema attraverso azioni codificate da una rom (BIOS --> UEFI) che contiene tutte le info per avviare un piccolo programma, contenuto nell'hardware, che permette alla macchina di identificare il sistema che vogliamo caricare
- (programma) identifica il dispositivo/supporto (memoria secondaria che contiene il S.O)
- (programma) esegue le istruzioni per caricare il file system, queste caricano le prime parti del sistema (prima fra tutti il kernel)
- Viene montato il disco principale

N.B.
Linux offre la possibilità di deviare il primo avvio con il dual boot: tramite un piccolo programma è possibile scegliere quale S.O. utilizzare (se entro 5 secondi non scegli un S.O. si avvia in automatico quello di default)

GESTIONE DELLA CPU E DEI PROCESSI (CAP 3)
Processo: programma in esecuzione = [statici] codice, dati + [dinamici] program counter, registri, stack, dati, risorse
Il sistema operativo gestisce l'ordine di esecuzione e la comunicazione dei processi.

Cosa succede se lancio lo stesso programma 3 volte? Ho tre istanze di processo distinte!

Rappresentazione interna del S.O.
Process control block (**PCB**): record con tutte le informazioni sul processo
Fondamentale sarà il numero del processo = Process Id (**PID**)

Quali informazioni necessita il sistema per gestire i processi?
- Gestione del processo
  - Program counter (dipende dall'architettura HW)
  - Registri (dipende dall'architettura HW)
  - Stato del processo
  - Priorità
  - …
- Gestione della memoria
  - Riferimenti a memoria --> conoscere allocazioni dedicate al processo
  - …
- Gestione I/O
  - Device assegnati
- Gestioni varie
  - File aperti
  - User ID (UID)
  - Group ID (GID)
  - Directory corrente (PWD)
  - Statistiche
  - …

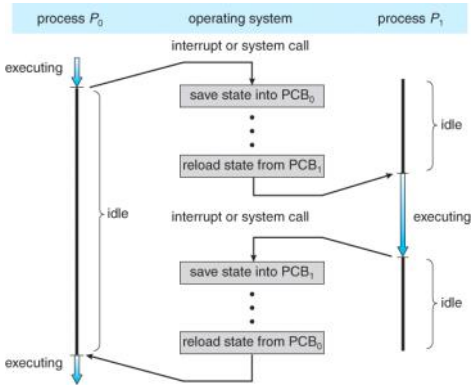
LEZIONE 13 (stati dei processi e scheduling)

Stato del processo
Cattura l'attività principale del processo in questo momento. Serve al S.O. per gestire l'evoluzione del processo stesso.

Stati della vita di un processo:

- NEW = il S.O. ha ricevuto la richiesta di creazione di un nuovo processo, ma questo non può ancora essere eseguito (assegno PID, non assegno risorse, creo PCB)
  "Ammissione al sistema" = il sistema alloca la memoria, predispone tutte le risorse che il processo necessita, aggancia le librerie necessarie, etc… , il processo può essere eseguito
- READY = il processo ha tutto quello che serve per poter essere eseguito tranne la CPU (sala d'attesa in cui si attende il processore)
  "Processore libero" = [scheduler a breve termine] posso scegliere quale processo andrà nello stato di esecuzione
- WAIT = il processo sta aspettando qualcosa
  - il processo rimane in coda wait finché non riceve la risorsa richiesta (es. apertura di un file, risposta dispositivo I/O, etc…)
  - Ottenuta la risorsa il sistema riceverà un interrupt e il processo tornerà nello stato di ready, pronto per essere eseguito
- EXEC = il processo lavora sulla cpu, ovvero la cpu esegue le istruzioni del processo tramite il suo program counter (max 1 processo)
  PRELAZIONE = [**Preamption**] il processo torna nello stato di ready; potrebbe capitare che il S.O. si rende conto che non è più conveniente far continuare l'esecuzione di quel processo (es. arriva un proceso con priorità più alta)
  "Ultima istruzione" = il processore esegue l'ultima istruzione del processo, il programma termina
- TERM = simmetrico allo stato di new
  - facciamo pulizia: il S.O. dealloca la memoria allocata, chiudiamo i file, rilasciamo le risorse, etc…
  - Il processo viene distrutto: perde il pid ed esce dallo stato di terminazione

Linea temporale:
Quando un processo esce dallo stato di EXEC, il S.O. interviene e salva lo stato del processo e copia le info nel PCB (il PC, i registri, etc..)
Quando il processo deve rientrare nello stato di EXEC, il S.O. carica lo stato del processo e legge dal PCB (il PC, i registri, etc..), dopodiché sposta il processo nell stato di EXEC



(CONTEXT SWITCH) COMMUTAZIONE DI CONTESTO (lezione 6.2 - minuto 15:25)
Attività di cambiare il contesto di esecuzione, in modo da gestire il cambio e ripristinare la situazione che un certo processo aveva lasciato.
Il S.O. cerca di dare l'illusione al processo che non sia successo niente nel periodo in cui non era in esecuzione sul processore.
Durante il context switch:

- Il processo non vede neanche di essere stato interrotto.
- Il sistema gestisce tutti i processi presenti e cerca di eseguirli tutti sul processore.

Per rendere tutto più efficiente, devo pianificare quali processi mandare sul processore prima di altri.

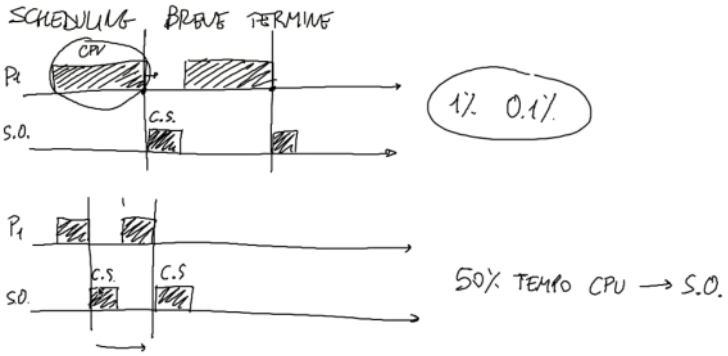
SCHEDULING - QUALI PROCESSI MANDARE IN ESECUZIONE
Lo scheduler deve garantire che tutti i processi vadano avanti, senza lasciare indietro nessuno.
Ho:

- Sistema multitasking (>1 processo eseguibile contemporaneamente)
- 1 processore
- Competizione per l'uso del processore (processi in coda ready)

Tipi di scheduling:

- **Scheduling a breve termine**: quale processo prelevare dalla coda ready per portarlo in esecuzione (~1000 volte al secondo)
- **Scheduling a lungo termine**: quale processo scegliere dalla coda new, al quale verranno assegnate tutte le risorse (~1 ogni minuto/ora)
- **Scheduling a medio termine**: gestisce il numero dei processi in esecuzione (grado di multitasking); quando ci sono troppi processi, ne sospendo alcuni temporaneamente, per evitare il sovrautilizzo delle risorse

SCHEDULING A BREVE TERMINE
Idea di base: prolungare il più possibile il tempo del processo in esecuzione per diminuire l'impatto del context switch, quindi evitare che 50% tempo cpu lo occupi al S.O. per le operazioni di context switch, ma piuttosto arrivare ad avere un rapporto dell'1% o meglio ancora dello 0,1%



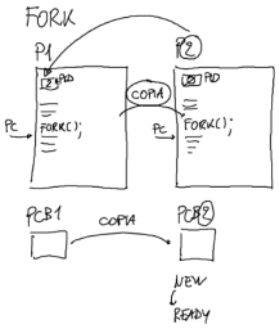
Coda di scheduling (struttura dati che ospita le informazioni necessarie)
- Coda ready: quale processo scelgo?
- Coda wait: aiuto il sistema ad identificare il motivo per cui sto aspettando
    - Evento I/O
    - Timer
    - Terminazione di un processo figlio
    - Interrupt

TIPOLOGIA DEI PROCESSI
- CPU Bound: passa la maggior parte del tempo sul processore (es. attività di calcolo, elaborazione di dati)
- I/O Bound: passa la maggior parte del tempo ad aspettare risposte da dispositivi di I/O (es. servizi, movimento cursore tramite mouse, editor di testo)

Come faccio a sapere la tipologia di un processo?
Statistica: misuro quanto un processo ha usato cpu rispetto al tempo di I/O = CPU / (tempo I/O + CPU) ---> ~100% CPU Bound / ~0% I/O Bound

LEZIONE 14 (operazioni sui processi)
OPERAZIONI SUI PROCESSI (CAP 3.3)
Analizziamo le operazioni in ambito Unix/Linux

Creazione
Un processo può creare un altro processo --> system call: Fork()
Si crea così una gerarchia di processi ad albero. Per ogni processo al suo interno possiamo conoscere:
- PID (Process ID) --> GETPID();
- PPID (Parent PID) --> es. GETPPID();



Quando facciamo una Frok() creaimo una copia identica del processo padre, per differenziarli, faccio così:
```
PID = Fork();
If(PID != 0) { } //significa che è il padre
Else { } //significa che è il figlio
```

System call usata per aprire un nuovo programma(da eseguire all'interno della sezione codice del figlio):
EXEC(path_programma): Butta via tutte le info copiate e sostituiscile con il programma che ti indico

N.B. La memoria non è condivisa!

Qual è il primo processo, quello alla radice di tutto?
In Unix/Linux viene chiamato INIT (PID = 1) ed è creato all'avvio della macchina; corrisponde a quel programma eseguito in partenza per permettere l'avvio del sistema

Terminazione
Uscita volontaria regolare: (lo zero sta ad indicare "tutto bene" --> il figlio termina dicendo al genitore ok)
- Return(0);
- Exit(0);

Uscita determinata dal genitore: (il genitore termina forzatamente il figlio)
- Segnale di terminazione che il genitore manda al figlio per incitargli di terminare
- Segnale di Kill, terminazione immediata (perdita parziale dei dati)

Uscita determinata dalla terminazione del genitore:
- il genitore finisce e uccide a cascata i figli
- Il processo init adotta i figli, in questo modo possono continuare la loro esecuzione

Errore (imprevedibile, operazioni illecite, operazione su file inesistente, etc…)

LIMITI DEI PROCESSI
- Condividono il processore (context switch)
- Non condividono la memeria
- Non condividono le risorse

Idea: considerare all'interno di un singolo processo la possibilità di eseguire dei sottoprogrammi in maniera indipendente o parallela = thread

THREAD (CAP 4)
Sceduling dei thread = scheduling dei processi

Vantaggi:
- Schema leggero
  - context switch è molto più pesante, devo copiare molte informazioni
  - mentre per il thread switch basta cambiare program counter e valore di registri, stack
- Tempo di risposta basso
  - Possono appoggiarsi a più core contemporaneamente: se abbiamo un thread che si blocca, non è un problema perché avremo altri core per mandare avanti gli altri thread
- Condivisione risorse
  - Rischioso, c'è il rischio che i vari thread facciano danni sovrascrivendo gli stessi registri
- Sfruttare a pieno architetture multicore (MIMD)

GESTITI DAL SISTEMA OPERATIVO
Dobbiamo informare il S.O. della presenza dei thread, in questa maniera il sistema sa quanti thread ci sono e come allocarli ai vari core (il kernel deve esser informato)

Creazione thread: tramite una system call

Libreria:
PTHREAD (standard Posix): è in grado di informare il kernel della presenza dei thread o se il S.O. non ha la visibilità dei singoli kernel, si occuperà simula una rotazione dei singoli kernel per il singolo core

Modelli multithread
- Modello uno a uno
  Per ogni thread informo il kernel della sua presenza, in modo che il SO possa decidere di mappare certi thread su certi core

- Modello molti a uno
  Mappare tutti i thread di un processo ad un unico progetto del kernel che verrà poi mappato su un singolo core (sconsigliato, si perde la possiblità di far lavorare i thread in parallelo sui vari core a disposizione)

N.B.
Su una macchina multiprocesso non è possibile che il processore esegua un thread di un processo e un thread di un altro processo;
Perché tutte le richieste affidate ad un certo processo avranno una certa porzione di memoria, quindi quando si fa il context switch sarà quel processo che genererà le sue richiesta alla ram, altrimenti mischieremmo le risorse.
Si può dare il processore per un po' di tempo al primo processo e mappare i vari thread sui vari core, dopo un po' si darà spazio al secondo processo con i suoi thread.
Se si hanno due processori invece è possibile eseguire su ognuno un processo, quindi si avrebbero due processi che lavorano contemporaneamente

SCHEDULING DEL PROCESSORE (CAP 5)
Analisi delle attività del processo: (utili per poterli poi classificare)
- BURST = tempo ininterrotto di uso uniforme di una certa risorsa (lunghezza dell'attività)
  - Burst di CPU = differenza tra quando esco e quando entro nello stato di esecuzione (t_uscita - t_entrata = burst_cpu)
  - Burst di I/O = differenza tra quando esco e quando entro nello stato di wait
- Comportamento del processo
  Misuro tutti i burst in un certo tempo; mi creo un elenco di burst di cpu e faccio una statistica per capire il comportamento di quel processo; creo un istogramma con asse x =lunghezza del burst e asse y = frequenza di burst

SCHDEULING DELLA CPU A BREVE TERMINE
Lo scheduler a breve termine sceglie il candidato ideale e lo porta in esecuzione.
Distinzione tra gli algoritmi:
- Con prelazione: il sistema ha la possibilità di portare un processo da exec a ready senza che quel processo abbia terminato, per permettere ad altri con più priorità di essere eseguiti (es. Windows 95)
- Senza prelazione (es. Windows 3.x)

Dispatcher (S.O. superutente)
Componente che da il controllo al sistema, si preoccupa di dare effettivamente l'avvio al processo che è stato selezionato:
- Gestire il context switch (salva PCB_uscita, carica PCB_entrata)
- Passare alla modalità utente
- Settare il program counter corretto del processo per lasciare il controllo

LEZIONE 15 (criteri di scheduling)

[criteri di scheduling] = utilizzati per ottimizzare l'esecuzione dei processi

Indici di prestazione:

- utilizzo della cpu:
  - misuro tutti i burst della cpu e ottengo un tempo
  - calcolo una percentuale di utilizzo della cpu basata su questo tempo in rapporto alla finestra di analisi
    - (finestra di analisi): comportamento del processo nel tempo, quando e per quanto tempo occupa la cpu
- frequenza di completamento(throughput): numero di processi completati nell'unità di tempo (es. 10 job al minuto)
- tempo di completamento (turnaround time): quanto tempo è passato dall'inizio fino alla fine del processo; misura non stabile e che non dipende solo dal processo:
  - tempo in coda ready
  - tempo attesa I/O
  - conta anche l'hardware, migliore = +veloce
- tempo di attesa (waiting time)(t.a.): somma di tutti i tempi passati in coda ready, attengo la CPU (processo pronto ma cpu occupata)
  - questo mi fa capire la capacità del sistema operativo nel smaltire le richieste di uso della cpu
  - t.a. basso = s.o. scarico + buona gestione
  - t.a. alto = s.o. carico + cattiva gestione
  - In un sistema interattivo ci aspettiamo che un evento abbia un certo effetto nel tempo (es. evento = schiaccio un tasto sulla tastiera, effetto = mi viene mostrato il carattere; es. evento = doppio click su exe, effetto = si apre una finestra <-- tempo + lungo)
- tempo di risposta = tempo fra l'inizio della risposta dalla richiesta

OTTIMIZZAZIONE

Dipende dal sistema in uso(server vs pc vs altro)
es. voglio tempi di risposta bassi, come faccio?

- calcolo la media dei tempi di risposta e mi assicuro che sia bassa, potrebbero esserci rari casi in cui il tempo è alto (non affidabile)
- calcolo la varianza e mi assicuro che sia bassa = poco discostamento da caso medio (affidabile)

LEZIONE 16 – 17 (Algoritmi di scheduling)

Algoritmi di scheduling:

FIFO

Supponiamo di avere 3 processi: p1, p2, p3 e il loro tempo di burst CPU (ipotetico, non si ha nella realtà).

esercizio 1:

ordine processi: p3-3ms, p2-3ms, p1-24ms

- tempo di burst: p1-24ms p2-3ms p3-3ms
- tempo di fine: p1-24ms p2-27ms p3-30ms
- tempo di attesa: p1-(0ms in coda ready) p2-(24ms in coda ready) p3-(27ms in coda ready)
- tempo di attesa medio = (0+24+27)/3 = 17ms <-- alto

esercizio 2:

ordine processi: p3-3ms, p2-3ms, p1-24ms

- tempo di burst: p1-24ms p2-3ms p3-3ms
- tempo di fine: p1-30ms p2-6ms p3-3ms
- tempo di attesa: p1-(6ms in coda ready) p2-(3ms in coda ready) p3-(0ms in coda ready)
- tempo di attesa medio = (6+3+0)/3 = 3ms <-- basso

Problema:

EFFETTO CONVOGLIO: c'è un processo pesante (cpu burst alto)(solitamente questi sono cpu bound) davanti che fa aspettare gli altri più leggeri (cpu burst basso)(solitamente questi sono I/O bound)

Riassumendo:

- non garantisce t.a. medio ottimo
- effetto convoglio
- risorse I/O poco sfruttate
- + efficiente

SHORTEST JOB FIRST (shortest next cpu-burst first [versione moderna])

/* job: attività che viene eseguita nel s.o. in modo ininterrotto
tempo di attesa: tempo in cui i processi aspettano in coda ready
esempio: lezione 17 - min 11 */

esercizio [SJF] :

- processi + burst: p1-6ms p2-8ms p3-7ms p4-3ms
- ordine exec + tempo fine: p4-3ms p1-9ms p3-16ms p2-24ms
- tempo di attesa: p4-0ms p1-3ms p3-9ms p2-16ms
- tempo medio di attesa: 7ms

esercizio [FIFO] :

- processi + burst: p1-6ms p2-8ms p3-7ms p4-3ms
- ordine exec + tempo fine: p1-6ms p2-14ms p3-21ms p4-24ms
- tempo di attesa: p1-0ms p2-6ms p3-14ms p4-21ms
- tempo medio di attesa: 10,25ms

Caratteristiche:

PRO → t.a. medio ottimo -> non si può fare di meglio

CONTRO → si basa sulla conoscenza del burst cpu, MA nessuno la conosce! Non è realizzabile!

Soluzione: fare delle previsioni

Quindi dobbiamo:
- dimostrare l'ottimalità dell'algoritmo
- approssimare l'algoritmo, ovvero stimare il burst

Riassumendo:
- *Scopo:* scelgo il prossimo processo con il burst minimo
- Aspetto algoritmico:
  - costo inserimento in coda ~O(logn)
  - costo estrazione dalla coda ~O(1)
- Dobbiamo ordinare dal processo con il burst più piccolo fino a quello con il burst maggiore per migliorare il tempo medio di attesa

Problema: Non conosciamo il tempo di burst!!
Soluzione: Guardo il passato / prevedo il futuro

Come? Memorizzo la storia dei processi, se la durata nel tempo è stabile allora ipotizzo una durata media futura... ma dovrei avare un array per la storia di ogni processo, è troppo dispendioso!

Quindi: faccio una media esponenziale (2 dati, 2 moltiplicazioni, 1 somma)
Calcolo: ▪ misura ennesima = misura del burst nel tempo n ▪ previsione ennesima = previsione per il tempo n ▪ previsione prossima (n+1) = ultima misura burst + ultima previsione (per maggiori dettagli guarda lez. 17 min. 37)

CRITERI DI SCHEDULING [OLD LESSON]

Indici misurabili che possiamo monitorare per prendere le decisioni, per poter pianificare i processi e quindi ottimizzare il sistema tramite degli algoritmi efficienti

Indici
- Utlizzo di cpu: (tempo di uso cpu / tempo di osservazione) = quanto tempo il processore è stato effettivamente usato (tempo in cui qualcuno è in exec) (sommatoria dei tempi di burst / tempo di osservazione)
  100% non raggiungibile da parte di un processo utente, una piccola parte è sempre dedicata al sistema operativo (es. context switch) [<1%]
- Frequenza di completamento (throughput): (numero job / tempo) //normalmente non indicativa perché catturiamo un'attività globale, tantissimi altri fattori influiscono sul valore
- Tempo di completamento (turnaround time): tempo di fine - (tempo di inizio) //diverso se testato più volte, consigliato avere più valori e fare una media/statistica
- Tempo di attesa (wainting time): tempo specifico che un determinato processo ha trascorso nella coda ready (il sistema sta facendo altro) = tempo sprecato //se c'è un solo processo nel sistema il tempo di attesa è zero
- Tempo di risposta (processo interattivo): quanto ci mette un processo a rispondere ad un certo stimolo (es. da quando premo un tasto alla risposta, es. carattere disegnato sullo schermo) = (tempo in cui inizio a vedere la risposta - tempo ottenimento richiesta)

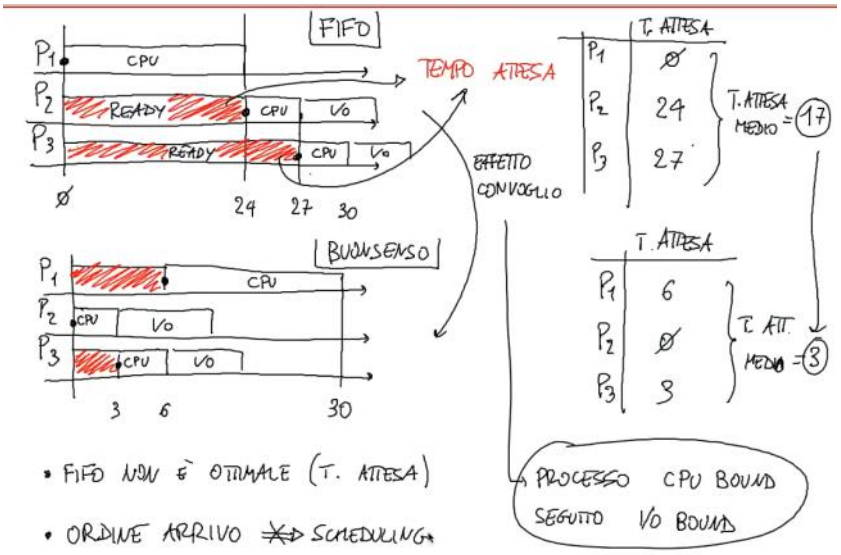
PCB <-- statistiche
OBIETTIVO: ottimizzare uno o più indici (es. minimizzare il tempo di attesa) --> combinazione di indice (es. media)

Combinazioni di indici:
- Minimizzo la media dell'indice: mediamente va bene ma in certi casi va molto male --> non prevedibile!
- Minimizzo la varianza dell'indice: cerco di far si che tutte le mie esecuzioni si comportino in modo uniforme; varianza estremamete contenuta; non prevedo una media bassa ma non prevedo una situazione in cui ho rendimenti molto diversi da quelli già visti --> sistema affidabile, molto robusto e prevedibile!

ALGORITMI DI SCHEDULING

Qual è il prossimo candidato da mandare in esecuzione?
Esempio:
Tempi di burst cpu: P1: 24ms, P2: 3ms, P3: 3ms

Ipotizziamo un algoritmo di scheduling che in base a queste statistiche inizia a chiedersi qual è il prossimo candidato da mandare in esecuzione?

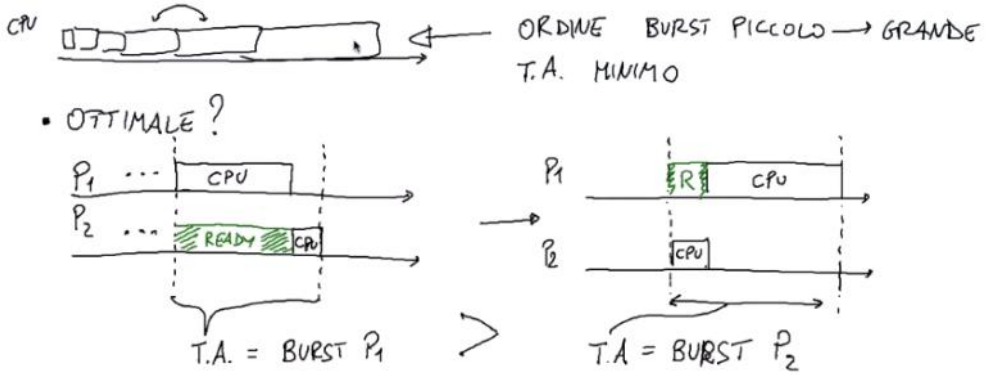


[ALG] First in first out (FIFO)
Il primo ad entrare in coda ready è il primo ad andare in exec
Pro: (efficiente: costo O(1)) tutti i porcessi vengono eseguiti
Contro: non si preoccupa dei tempi di attesa = **EFFETTO CONVOGLIO**: processo estremamente pesante seguito da tanti processi dietro di lui

[ALG] Algoritmo "buonsenso"
Prevede di far passare prima i processi più leggeri.
Pro: tempo di attesa minimo
Contro: il processo pesante potrebbe non esser mai eseguito

[ALG] Shortest job first (SJF)
SJF --> Shortest next cpu-burst first (nome corretto + recente)
L'algoritmo "buonsenso" in realtà esiste ed è SJF (N.B. Job = stima del burst cpu)
Ordino i processi da quelli con burst cpu minimo a quelli con burst cpu massimo, estraggo il minimo e lo porto in exec
Pro: minimizza al massimo il tempo di attesa medio
Contro: rischio che un processo pesante non venga mai eseguito e che quindi abbia un tempo di attesa "infinito" = **STARVATION**
inoltre
Problemi:
1) Siamo certi che minimizza al massimo il tempo di attesa, non si può fare di meglio
2) Non conosciamo il burst di cpu!

Problema 1:
Possiamo dimostrare che è ottimale: minimizza al massimo il tempo di attesa



Problema 2:
Non conosco il burst di cpu, ma posso sfruttare le statistiche del processo
- cpu bound --> burst cpu alto
- I/O bound --> burst cpu basso

Sfrutto quindi la storia dei burst per "predirre il futuro"

PRINCIPIO DI LOCALITA'
Qualcosa che è successo di recente nel passato è molto probabile che succeda nel futuro prossimo;
qualcosa che è successo nel passato molto remoto non ci può più dare molta previsione sul futuro;
qualcosa che sta succedendo in questo momento è possibile che succeda ancora

Come possiamo fare a tenere questa memoria delle statistiche del processo?

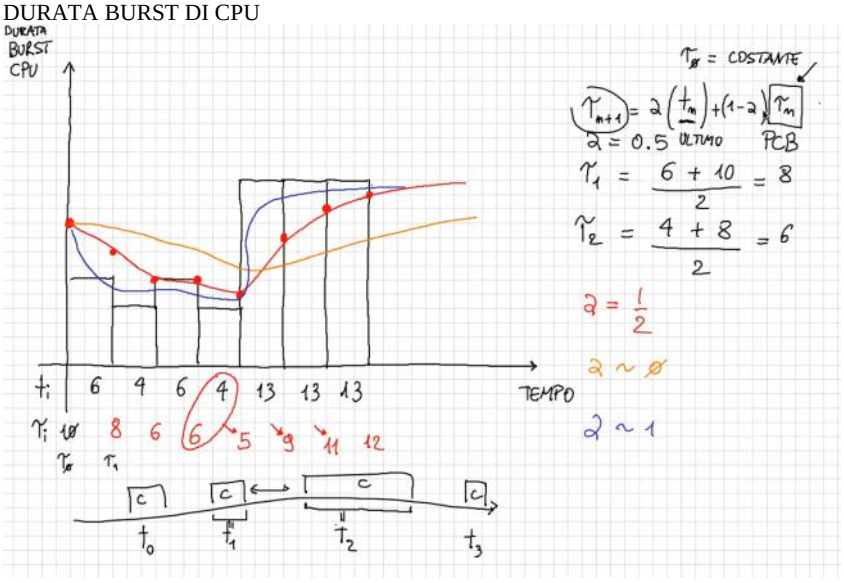
SCHEDULING A MEDIA ESPONENZIALE
Previsione basata sulla storia, media aritmentica a tutti gli effetti che viene memorizzata nel PCB e viene aggiornata ogni volta che c'è un nuovo burst
Usata come stimatore del futuro ma in continuo adattamenteo alle nuove situazioni.

N.B. Esponenziale: ogni volta che vado indietro nel passato considero un burst esponenzialmente più basso
T(n+1) = 1/2*t(n) + 1/2*(1/2*t(n-1) + 1/2*T(n-1) = (1/2)^1*t(n) + (1/2)^2*t(n-1) + (1/2)^3*t(n-2) + (1/2)^4*t(n-3) + …

Come funziona:
Dati:
- t(i) = Burst cpu i-esimo //quello che davvero è successo, ultimo burst misurato
- T(j) = Previsione del burst i-esimo //quello che pensavo sarebbe successo, ultima previsione
- T(n+1) = α*t(n) + (1-α)*T(n)
  Dove α ϵ (0 .. 1) //controlla quanto la previsione si fa infleunzare

Esempio:
α = 1/2 --> T(n+1) = (t(n) + T(n))/2 //influenza moderata
α = 0 --> T(n+1) = T(n) //influenza poco
α = 1 --> T(n+1) = t(n) //influenza molto

DURATA BURST DI CPU



CONCLUSIONE
Utilizzando l'algoritmo SJF + media esponenziale approssimo i burst futuri, quindi faccio le mie previsioni e scelgo come prossimo processo da mandare in exec quello con la previsione più bassa.

LEZIONE 18 (scheduling a priorità)

(se il tempo di attesa medio non è l'indice a cui siamo interessati)

SCHEDULING A PRIORITA'

Algoritmo simile a SJF ma sceglie prima i processi con *priorità* più alta, non quelli con burst più basso

Come calcolare la priorità?
- Priorità interna(base): calcolata dal s.o. su indici del processo, tipo: burst cpu, limiti di tempo, richieste di risorse(memoria, etc.), tempo I/O, tempo cpu
- Priorità esterna(modificata): decisa da un amministratore/utente

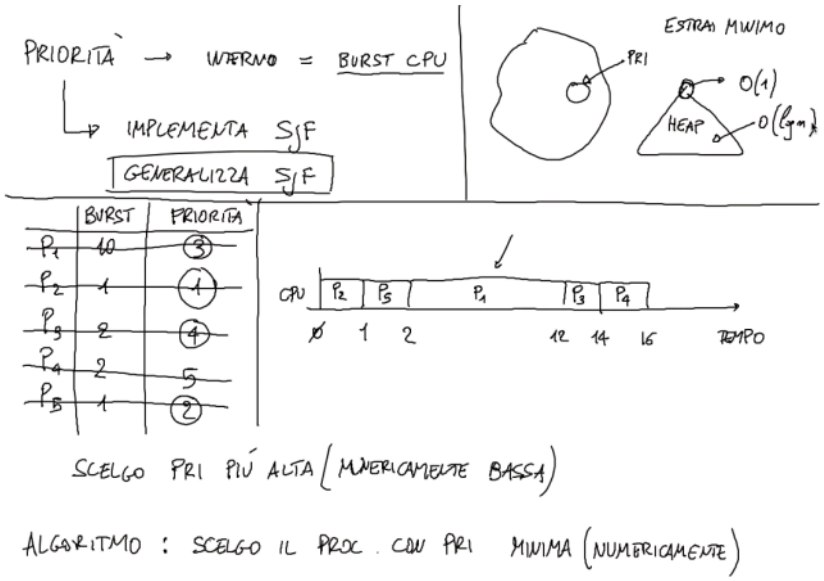
PRELAZIONE:
Cosa succede: un processo in exec viene spostato/trasferito in coda ready per far spazio ad un altro processo
Quando succede: quando un nuovo processo entra in coda ready e ha una priorità maggiore di quello in exec
Utilità: da la possibilità a dei processi con priorità maggiore di essere eseguiti subito
Problema:
STARVATION = processi a priorità bassa rimangono sempre in coda ready perchè ne arrivano sempre di nuovi con priorità maggiore
Soluzione:
AGING = più un processo permane in coda ready più la sua priorità incrementa

/****** OLD LESSON ******/

SCHEDULING A PRIORITA'
Tipologie:
- Interna (variabile): nasce da misure di comportamento del nostro processo all'interno del sistema (es. uso delle risorse, limiti di tempo, IO time/CPU time, …)
- Esterna (fissa): priorità insindacabili stabilite in precedenza e che il S.O. non può controllare (es. l'amministratore può avere più uso del processore rispetto ad altri utenti)

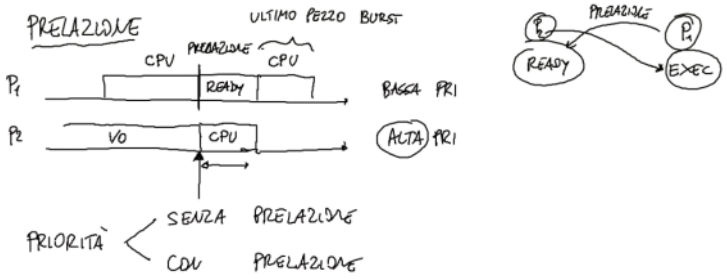
Priorità = intero 0 .. K (dove 0 = 0 massima e k = minima)

Esempi:
- [SJF con scheduling a priorità] (lezione 9.2 min ~20)
  Indice interno = burst di cpu //chi ha un burst cpu minore avrà più priorità



- [Prelazione] (lezione 9.2 min ~22)

  Scheduling a priorità:
  - Con prelazione: ho la possibilità di soddisfare più rapidamente tutte le richieste dei processi con priorità più alta
  - Senza prelazione: non ho " "



- [Starvation] (lezione 9.2 min ~25)
  Un processo a priorità bassa non viene mai scelto

  Soluzione: AGING = più un processo invecchia più la sua priorità aumenta

Quello che abbiamo visto finora non permette la realizzazione di un sistema interattivo!

LEZIONE 19 (Round Robin)

ROUND ROBIN

Algoritmo pensato per implementare lo scheduling time-sharing

TIME SHARING: tutti i processi hanno la stessa priorità e tutti i processi vengono eseguiti per una piccola fetta di tempo (Q = quanto di tempo); il tempo cpu viene suddiviso in parti uguali a tutti i processi

Vantaggio: posso creare un sistema interattivo (min. 7) + prelazione + tempi di risposta controllati

Quanto di tempo = Timer programmabile:
- un processo va in exec
- parte il timer
- poi ci sono 2 possibili evoluzioni:
  - il processo lascia exec, reset timer
  - scatta il timer, interrupt, reset timer, prelazione, ready

Domande:
- Ci può essere un effetto convoglio? NO → se ci sono n processi in ready, entro il tempo n*Q eseguo tutti i processi in ready

- Può esserci Starvation? NO → è impossibile che un processo non venga mai scelto
- N.B. La scelta di Q è molto importante!!!
    * Q alto: poco efficiente(Fifo)
    * Q basso: sembra ottimo ma c'è da considerare la fase di context switch (rischio di passare la maggior parte del tempo a copiare dati)
    * Q ottimo: solitamente 100 o 1000 volte il context switch

Riassumendo: (lez. 19 - min. 20)
- è un alg con prelazione, la prelazione avviene allo scattare del quanto di tempo associato al processo
- base a coda FIFO
- più attesa rispetto SJF, ma no effetto convoglio, no starvation

/****** OLD LESSON ******/

ROUND ROBIN (CODA CIRCOLARE + QUANTO DI TEMPO) [OLD LESSON]

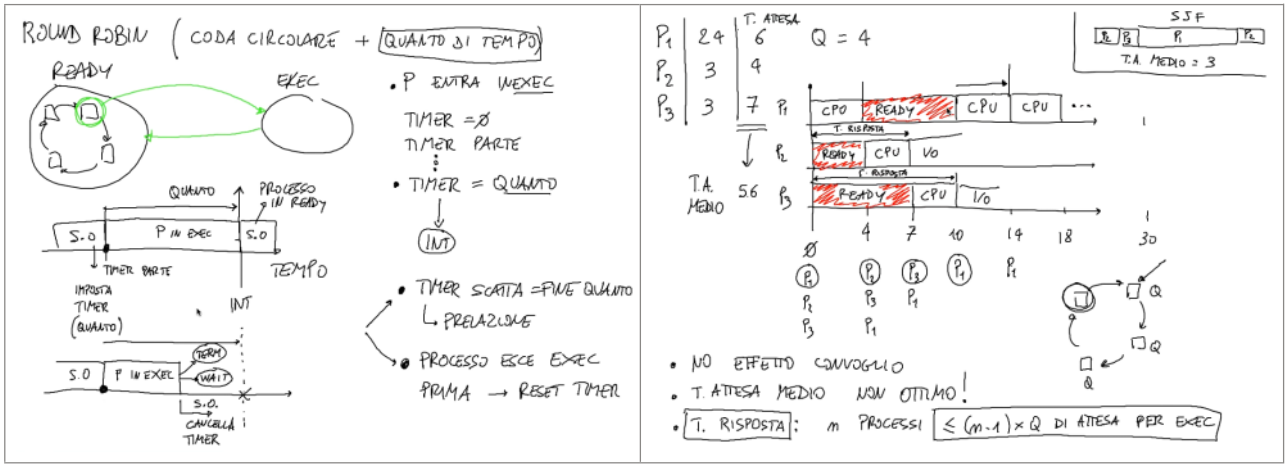
Come funziona:
- Il processo entra in exec, il SO:
    - resetta il timer --> timer = 0
    - fa partire il timer(timer = quanto)
- Timer = quanto (int)

Situazioni:
1) Timer scatta = fine quanto di tempo --> prelazione
2) Il processo esce prima dall'exec --> reset timer

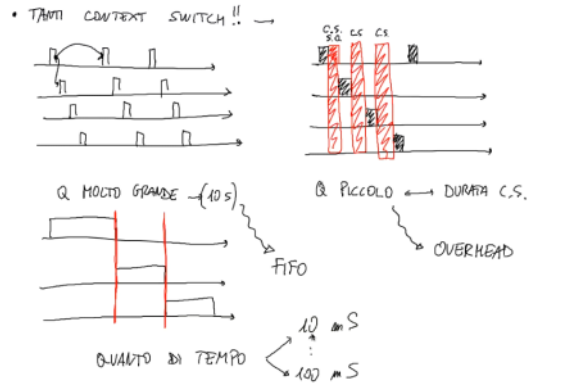
Pro:
- no effetto convoglio
- Tempo di risposta medio buono: n processi <= (n-1) * Q di attesa per exec

Contro:
- tempo di attesa medio non ottimo!
- Tanti context switch

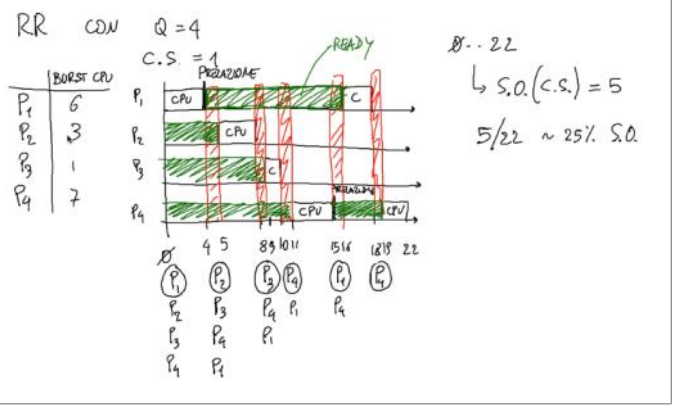
Quanto di tempo ottimale: da 10ms a 100ms



Esempio 1:



Esempio 2:

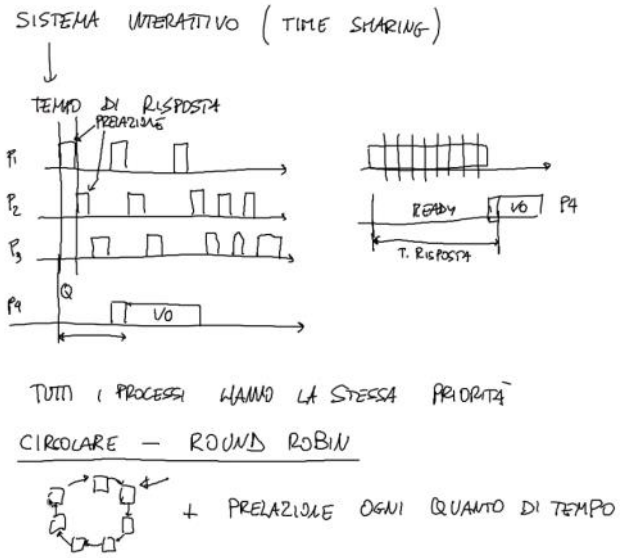


/****** OLD LESSON ******/

SCHEDULING A CODA CIRCOLARE (ROUND ROBIN) (lezione 9.2 min ~44)
SITEMA INTERATTIVO (Time sharing)

Introduco un tempo di risposta massimo in cui un processo può stare sul processore; il tempo di risposta è molto più anticipato

Prelazione: spezzetto il burst di cpu in tante fette e poi distruibuisco queste fette sul processore un po' alla volta
N.B. Tutti i processi hanno la stessa importanza



LEZIONE 20 (scheduling a più livelli)

SCHEDULING A PIU' LIVELLI (estensione alg. scheduling a priorità)

Le importanze dei vari processi sono caratterizzate su più livelli (quindi: livello 0, 1, ... , k-1)

Caratteristiche:
- organizzato in code, concetto di CODE MULTIPLE
- posso associare ad ogni coda un algoritmo di scheduling diverso

Esempio:
- coda livello 0: processi foreground --> RR Q=basso
- coda livello 1: processi background (minore priorità, no interattività) --> FIFO + prelazione

Come funziona?
- identifico una coda, prima coda non vuota partendo dalla coda più importante
- attivo un alg. di scheduling per quella coda

FUNZIONALITA' AGGIUNTIVE: al concetto di coda multipla si può aggiungere anche il concetto di retroazione.

RETROAZIONE (feedback): si può cambiare la coda di appartenenza di un processo; quando un processo rientra dallo stato di exec in coda ready posso decidere se farlo rientrare nella stessa coda o cambiare.

Perchè? ci siamo accorti che è più prioritario/importante di quanto pensavamo.

Quindi (nuova coda):
- +1 = coda più penalizzate (burst > Q)
- +0 = stessa coda
- -1 = coda più privilegiata (burst < Q)

PRO: il sistema si adatta ai bisogni di ogni processo

CONTRO: starvation

LEZIONE 21 (Scheduling Real Time)

SCHEDULING REAL TIME

- Hard Real Time: si chiede che un'operazione termini in un certo tempo (es. telefoni)
- Soft Real Time: processi critici --> associano priorità troppo alta, più del s.o. (es. portatili)

Cosa succede se arriva un processo con più priorità durante una system call(n.b. il s.o. ha la priorità maggiore):
- eredità temporanea della priorità da un processo con pri. bassa(quello che ha chiamato la system call) ad un processo con pri. alta (quello in arrivo)(min 9) <-- in poche parole, il processo con pri bassa eredità momentaneamente(per la durata della system call) la stessa pri del nuovo processo in coda di ready con pri più alta.
- identifico dei punti sicuri nel kernel in cui posso sospenderlo, in cui si può fare prelazione(PREEMPTION POINTS) (ma è diffice da gestire!) <-- in poche parole trovo dei punti in cui posso interrompere la system call, punti dove non avviene perdita di dati, da lì posso procedere con la prelazione, cioè sposto il processo A in coda di ready per mandare in exec il processo B.

Come fare per proteggere i dati?
- sincronizzazione
- comunicazione

LEZIONE 22 (Comunicazione)

I processi:

- possono essere indipendenti (cioè non condividono i dati con altri / esecuzione non influenzata da altri)
- possono essere cooperanti (cioè si scambiano dei dati o si influenzano)

Problema:
- il s.o. blocca ogni forma di interazione tra processi!

Perchè voglio avere più processi che comunicano?
- condivisione di informazione
- + parallelismo
- modularità

Tipologia di meccanismi per gestire questa comunicazione:
- sincronizzazione, influenzare la comunicazione e sospenderla se necessario in attesa di un altro processo
- scambio dei dati, tramite:
  - memoria condivisa
  - scambio di messaggi

MEMORIA CONDIVISA

Cosa succede nella memoria condivisa? (7.25 min)
- ci sono 2 processi paralleli, concorrenti, che condividono la stessa memoria (produttore e consumatore)
- 1 buffer, cioè uno spazio di appoggio che ci permette di tenere a mente i dati condivisi

Problemi?
- Sincronizzazione:
  - prod. può scrivere solo se c'è posto
  - consum. legge solo se il buffer non è vuoto, se è vuoto deve aspettare
- Dimensione del buffer:
  - capacità 0, cioè il buffer non esiste --> sincronizzazione necessaria, cons e prod si devono posizionare nello stesso istante di tempo per poter comunicare
  - capacità infinita, il prod non si ferma mai, non ha vincoli di sincronizzazione (situazione impossibile)
  - capacità limitata, il prod ha un limite

Come possiamo realizzare questo meccanismo? (14 min)

Supponiamo di avere un buffer circolare condiviso:
- puntatore IN, spazio da scrivere
- puntatore OUT, spazio da leggere

Possibili casi:
- buffer vuoto: IN = OUT
- buffer pieno: IN = OUT (ambiguità! -> per distinguere ci fermiamo una cella prima) → (IN+1)%n = OUT

esempio:

[variabili condivise]:
- int n
- int buffer[n]
- int IN=0
- int OUT=0

Produttore:
```
while((IN+1)%n == OUT){} //busy waiting, sincronizzazione non efficiente, perdo del tempo perchè devo aspettare dei cicli di clock
//se riesco a oltrepassare il while, significa che c'è almeno una cella che posso scrivere
buffer[IN] = dato
IN = (IN+1)%n
```
Consumatore:
```
while(IN == OUT){} //busy waiting
//superare la barriera vuol dire che il produttore ha prodotto almeno un elemento (IN > OUT)
dato = buffer[OUT]
OUT = (OUT+1)%n
```

Come realizzo la memoria condivisa?
- System call del s.o. per richiedere la memoria (IPC -> Inter Process Communication)
- Programmazione con i thread, flussi di exec all'interno dello stesso processo, quindi hanno la memoria condivisa "gratis"

LEZIONE 23 (MailBox)

SCAMBIO DI MESSAGGI

Vantaggio: comunicazione diretta: dichiaro esplicitamente chi deve mandare e chi deve ricevere

N°	ESEMPIO	PROBLEMA
1	A=send(B,mess); B=receive(A,mess) simmetrico	Voglio ricevere dei mess ma non so chi può mandarlo, es. telefono
2	A=send(B,mess); C=send(B,mess); B=receive(ID,mess) <~~ asimmetrico	Come faccio ad identificare ID, cioè il mittente?
3	[A=send(MBX,mess)] --> MailBox: condiviso attraverso s.o. --> [B=receive(MBX,mess)] <~~ comunicazione indiretta	Come faccio ad identificare ID, cioè il mittente?

LEZIONE 24 (Sincronizzazione)

SINCRONIZZAZIONE

Meccanismi offerti dal SO per risolvere questo problema:

RACE CONDITION(corsa critica): risultato imprevedibile, il risultato del calcolo dipende dall'ordine di esecuzione delle istruzioni (considerate a

basso livello) <-- non avviene nei programmi sequenziali

ESECUZIONE ATOMICA: non avviene il context switch

SEZIONE CRITICA: parte di codice che non può essere divisa da altre esecuzioni (più piccola possibile xk altrimenti perderei la concorrenza)

Dal punto di vista del s.o., il mio programma:
- esegue una parte non critica
- esordio, sys call s.o. per avvisare che entro in sez. critica
- sezione critica
- epilogo, sys call s.o. fine sez. critica
- sezione non critica

Proprietà:
- mutua esclusione: al massimo un processo alla volta si trova nella sua sezione critica
- attesa limitata: nel momento in cui un processo chiede di entrare in sez critica, esiste un numero di volte massimo prefissato in cui altri processi entrano nella loro sez critica (per garantire una equità nell'accesso)
- progresso: un processo uscendo dalla propria sez critica decide chi è il prossimo processo che deve andare in sez critica; in altre parole, solo i processi interessati ad entrare concordano chi entra effettivamente nella sez critica

LEZIONE 25 (sezione critica - Peterson)

Caratteristiche:

All'interno di una sez critica non vogliamo essere interrotti, quindi:
- deve essere atomica
- non vogliamo context switch

Implementazioni sezioni critica:

- *Disabilito gli interurpt*

Impedisco al sistema di attivare la richiesta dei context switch, evito di lanciare altri processi. La mia sezione criticà sarà:
- esordio = disabilito interrupt
- epilogo = abilito interrupt

Questa scelta è molto rischiosa, il s.o. potrebbe perdere il controllo, quindi:
- no utente: si nega questa possibilità al semplice utente
- sì kernel

Rispetta le proprietà?
- mutua esclusione: sì, nessun alrto può essere eseguito nel mentre
- attesa limitata: no, non è grarantito che un processo esca e rientra un numero limitato di volte
- progresso: sì, la scelta viene fatta da chi riesce a disabilitare gli interrupt per primo, che vincerà la competizione

- *Variabili di lock*
  - var = 0 --> sez critica vuota
  - var = 1 --> sez critica occupata

  esempio:

```
[sezione non critica]
while(var == 1) {} var=1 //esordio
[sezione critica]
var=0 //epillogo
[sezione non critica]
```

Rispetta le proprietà?
- race condition, se fatta da due processi contemporaneamente
- non c'è la mutua esclusione

- *Alternanza stretta*

Es. (01010... ok - 001... not ok)
Variabile di turno:
- turno 0 --> processo 0
- turno 1 --> processo 1
- Rispetta le proprietà?
- mutua esclusione: sì
- attesa limitata: sì
- progresso: no

- *Algoritmo di Peterson per 2 processi (Dijkstra 65')*

Variabili condivise:
- int turno = 0
- int flag[2] = {0,0} //descrivono l'interesse ad entrare

esempio:

```
// P0
    flag[0]=1
    turn=1
    while(flag[1]==1 && turn==1){}
    [sezione critica]
    flag[0]=0
// P1
```

flag[1]=1
turn=0
while(flag[0]==1 && turn==0){} //busy waiting
[sezione critica]
flag[1]=0

Rispetta le proprietà?
- mutua esclusione: sì
- attesa limitata: sì
- progresso: sì

Svantaggio:
BUSY WAITING: durante una sez critica un altro processo aspetta di poter entrare nella propria, stiamo usando la cpu per aspettare!

LEZIONE 26 (algoritmo baker)

- *Algoritmo Baker* (estesione alg. Peterson)

Distributore biglietti numero supermercato, variabili:
- Num[n] = {0, … , 0} //biglietti (es. num[i] = biglietto di i)
- Flag[n] = {0, … , 0} //interesse a prendere il biglietto

esempio:
```
flag[i] = 1 //ho interesse a prendere il biglietto
num[i] = 1 + MAXj{ num[j] } //conto tutti i biglietti presi e prendo il primo libero
flag[i] = 0 //non ho più interesse perché ho già il biglietto
for j=0 → n-1
      while ( flag[i] == 1 ) {} //se j (altro processo) sta ancora prendendo il numero, devo aspettare
    * while ( num[j] != 0 && num[j] < num[i]) {} // j (altro processo) ha un numero valido? j è prima o dopo di me? Se è prima devo aspettare
num[i] = 0 //lascio il biglietto
```

- Race condition

Problema: è possibile che dopo l'attribuzione dei numeri ci siano dei processi con lo stesso numero
Soluzione: [modifica] *
```
while ( num[j] != 0 && (num[j], j) < (num[i], i)) {} //guardo un altro parametro, per esempio il PID; quindi, nel caso due processi abbiano lo stesso numero, do io un ordine basato sul loro PID pubblico
```

Rispetta le proprietà?
- Mutua esclusione: 2 processi
- Attesa limitata
- Progresso

LEZIONE 27 (HW di sincronizzazione)

Abbiamo già visto con le var di lock che è una buona idea avere una variabile che ci descrive se una sezione critica è occupata o libera

Problema: race condition che ci impedisce di entrare

Soluzione:

- *Teorica*

  [Esordio]
  while (var==1) {}
  var=1

- *Pratica*

  E' piena di race condition, perché continuo a fare un get e successivamente un set
  Riesco a rendere l'attività atomica? (In questo modo potrei fare entrambe le operazioni insieme)

  A livello Hardware è stata introdotta l'operazione TEST&SET
  Cosa avviene? Viene caricato su un registro il valore della mia variabile (TSL R0, var)
  - R0 ← var
  - var = 1
    - [Esordio]
  - TSL R0, var
  - CMP R0, 0
  - JNE
  - sezione critica
  - var = 0

Riespetta le proprietà?
- Mutua esclusione: sì, avendo le operazioni atomiche non c'è la possibilità che due progressi entrino simultanemente nella propria sez critica
- Progresso: si, solo chi vuole entrare in competizione riuscirà ad avere il lock a 1
- Attesa limitata: no, non c'è nessuna garanzia che un processo entri un certo numero di volte lasciando un altro sempre ad aspettare

LEZIONE 28 (busy waiting – sleep/wakeUp)

Problema: utilizziamo cicli di clock per perdere tempo!
Soluzione: migrazione dallo stato di exec a quello di wait

Ipotesi:
- sys call → Sleep() //da exec a wait
- sys call → wakeUp() // da wait a ready

Esempio 1:

Algoritmo di Peterson a priorità

Processi:
- A (priorità alta)
- B (priorità bassa)

B entra in sez critica → ready
A → exec → busy waiting → ready
Scheduler → A //ma A aspetta B, quindi busy waiting infinito

Esempio 2:
CT = 0
- *Produttore:*
  if(ct == n) sleep() //possiamo volontariamente addormentarci al momento opportuno
  add(dato)
  ct++
  if(ct==1) wakeUp(consumatore) //ma dobbiamo preoccuparci di svegliare qualcun altro
- *Consumatore:*
  if(ct==0) sleep()
  leggi()
  ct--
  if(ct==n-1) wakeUp(produttore)

Problema:
Combinazione sfortunata di sleep()/wakeUp() tramite context switch al momento sbagliato
Soluzione:
Contatore di sveglie: se anche arrivasse un wakeup prima della mia sleep, in qualche modo ho un credito, so che mi devo svegliare, quindi se provo ad addormentarmi ecco ceh la mia sveglia me lo ricorda e mi consuma il credito

LEZIONE 29 – 30 (semafori)
Semaforo: generalizzaione concetto sleep/wakeUp, risolve il problema di sveglia persa

Può essere visto come un contatore:
- int contatore = 0 (privato) //contatore di sveglie
- down()
- up()

Cosa avviene:
- potrei ricevere alcune sveglie nel tempo e potrei acucmularle;
- ogni volta che faccio una up() incremento il contatore
- ogni volta che faccio down() decremento il contatore
- se arriva una down() wuando il contatore è 0, allora dobbiamo addormentarci → sleep()
- se siamo addormentati e riceviamo una up(), allora ci svegliamo → wakeUp()

Esempio 1:

```
Down()
if(contatore==0) sleep()
else contatore—
Up()
if(qualcuno dorme) wakeUp()
else contatore++

Semaforo Mutex (Mutual Exclusion) → sem(1) [down() … up()] → al max un processo si trova nel mezzo
```

[Semaforo Mutex] (min 14:40)

P0	P1
Sem.down() → cont = 0	…
Sezione critica	Sem.down() Sleep()
Sem.up()	wakeUp()
…	Sezione critica
…	Sem.up() → cont = 1

Concettualmente (dal punto di vista del consumatore):
- Il consumatore si deve fermare quando il buffer è vuoto
- MA noi potremmo nascondere all'interno del semaforo il concetto di quanti elementi ci siano nel mio buffer
- ALLORA potrei iniziare inizializzando un semaforo a 0 → pieni(0) //questo può essere usato dal consumatore per fare un semplice controllo
  - Pieni.down() //il cons sta per consumare, se ho 0 elementi il cons deve fermarsi, altrimenti posso andare avanti
- A questo punto dovrò anche mandare un segnale al produttore, magari perché è sospeso
- Inizializzo un semaforo che conta il numero di elementi vuoti del buffer → vuoti(0)
  - Vuoti.up()
- Per essere più precisi potremmo aggiungere un semaforo di mutex(1) per proteggere l a sezione critica per l'attività di consumare/produrre

Rispetta le proprietà?
- Mutua esclusione: sì, se il semaforo ha valore 1 almeno uno dei due processi si sarà addormentato
- Attesa limitata: sì, il fatto che un processo faccia down → sleep potrebbe dare la possibilità ad un altro processo di entrare ed uscire dalla sua s.c. tante volte, ma questo non accade perché il modo in cui inseriamo i processi nella coda wait viene gestito come una coda, quindi ci

permette di addormentare i vari processi sul semaforo senza che ci siano degli scavalcamenti nella coda, quindi è impossibile che un processo rientri nella s.c. senza anver risvegliaot prima qualcun'altro
- Progresso: sì, solo quelli che vogliono fare la down saranno in competizione per vincere il credito del semaforo, tutti gli altri troveranno il semaforo a zero e si addormenteranno

Spiegazione per babbei:
N.B. MUTEX = al massimo un'auto alla volta può attraversare e occupare l'incrocio!

Caratteristiche:
- Ci sono due auto (P0 e P1) che vogliono attraversare un incrocio
- L'incrocio è regolato da un semaforo e solo un'auto alla volta può attraversare

Risorse:
- P0, P1 = auto
- Sem(1) = semaforo con rispettivo stato
- Contatore = stato del semaforo
  - 0 → qualcuno sta attraversando l'incrocio (sem rosso)
  - 1 → nessuno sta attraversando l'incrocio (sem verde)

Azioni:
- S.C (sezione critica) = attravaersamento da parte di un'auto
- Sem.Down() = chiedo al semaforo di accendersi per poter attraversare
- Sem.Up() = dico al semaforo che può spegnersi perché ho attraversato
- Sleep() = qualcuno sta già attraversando, aspetto
- WakeUp() = chi stava attraversando ha concluso la manovra, ora posso attraversare

Esempio 2:
[Consumatore-Produttore] (min 25:00)

Consumatore	Produttore
pieni.down() mutex.down() consuma mutex.up() vuoti.up()	vuoti.down() mutex.down() produce mutex.up() pieni.up()

Rispetta le proprietà?
- Mutua esclusione
- Attesa limitata
- Progresso

Spiegazione per babbei
Caratteristiche:
- In un ristorante ho a disposizione un certo numero di piatti
- Ogni voolta che si svuota un piatto, uno dei cuochi può nuovamente cucinarlo
- Ogni volta che si riempi un piatto, uno dei clienti può mangiarlo

Risorse:
- Consumatore = cliente
- Produttore = cuoco
- Pieni(0) = semaforo che rappresenta lo stato dei piatti pieni
- Vuoti(0) = semaforo che rappresenta lo stato dei piatti vuoti
- Mutex(1) = semaforo

Azioni:
- Consuma() = [s.c. consumatore] il cliente sta mangiando
- Produce() = [s.c. produttore] il cuoco sta cucinando
- Pieni.down() = [cons.] se c'è un piatto pieno, lo mangio
- Pieni.up() = [prod.] ho cucinato un piatto, adesso è pieno
- Vuoti.down() = [cons.] se c'è un piatto vuoto, lo cucino
- Vuoti.up() = [cons.] ho mangiato un piatto, adesso è vuoto

LEZIONE 31 (monitor / 5 Filosofi)
- monitor -
Implementa un meccanismo di sincronizzazione più ad alto livello rispetto ai semafori.
Dari:
- Offre delle variabili di condizione su cui è possibile fare le richieste di sincronizzazione
- Chiamate di wait e signal che seriviranno per fare le richieste di sincronizzazione (diverse da up e down dei semafori)

Meccanismo:
- Wait(var condizione) //sospende il processo
- Signal(var condizione) //risveglia un processo sospeso

Differenza sostanziale rispetto ai semafori:
Il monitor assomiglia ad una classe, sono specificati una serie di metodi. La particolarità sta nel fatto che eseguendo i metodi esegue in mutua esclusione rispetto agli altri metodi, ovvero è l'implementazione stessa del monitor che ci garantirà la proprietà di mutua esclusione: ci semplifica la vita, quando esegue un metodo so per certo che è eseguito in mutua esclusione rispetto ad altre possibili chiamate di altri processi. Tuttavia potremmo comunque doverci sincronizzare con gli altri processi, per farlo entrano in gioco wait e signal. Una volta che una wait è chiamata all'interno di un metodo non solo sospendiamo il processo, ma rilasciamo anche il monitor, ovvero diamo la possibilità ad altri processi di entrare nei metodi del monitor.

Con questo meccanismo garantiamo che il protocollo funzioni correttamente.

Esempio (produttore e consumatore):

```
var_condizione pieni, vuoti;
int ct=0;
Void Inserisci(int item){
        If(ct==n) wait(pieni); //il processo si addormenta e rilascia il monitor, segnalo al monitor che non sono più in exec
        //quando veniamo risvegliati
        Add(item);
        Ct++;
        If(ct==1) signal(vuoti) //dobbiamo ricordarci di risvegliare il consumatore
}
Int Estrai(){
        If(ct==0) wait(vuoti);
        ret = read();
        Ct--;
        If(ct==n-1) signal(pieni) //il processo riprende la sua esecuzione
}

/*****************************************************************************/

Prod --> monitor.inserisci(item)
Cons --> monitor.estrai()
```

Tutto il codice è eseguito automaticamente in mutua esclusione, non c'è bisogno di creare un semaforo mutex per prooteggere le attività.
Non è una buona idea mettere un codice enorme all'interno della funzione di un monitor, perché in questo caso ridurremmo il parallelismo; quindi bisogna limitare le chiamate al monitor per le attività essenziali.

- 5 filosofi -
Problema: race condition
Soluzione: utilizzo di semafori

Filosofo:

Filosofo(i)

while(T){
Pensa()
Prendi(i)
Predni(i+1)%5
Mangia()
Posa(i+1)%5
Posa(i)
}

RICORDA

Down()
if(contatore==0) sleep()
else contatore—
Up()
if(qualcuno dorme) wakeUp()
else contatore++

Semaforo Mutex (Mutual Exclusion) → sem(1) [down() … up()] → al max un processo si trova nel mezzo

Algoritmo 1
Semaforo per bacchetta

Sem_i (1)
Prendi(i) Sem_i.down() Posa(i) Sem_i.up()

Problema: DEADLOCK (stallo) → blocco totale per via di una particolare sincronizzazione, tutti i filosofi prendono la bacchetta alla loro destra e rimangono in attesa di prendere la bacchetta sinistra
Soluzione 1: filosofo mancino, uno dei filosofi prende prima la bacchetta sinistra anziché la destra, questo fa sì che almeno un filosofo compierà l'azione e riuscirà a sbloccare gli altri
Soluzione 2: filosofo timido, uno dei filosofi prova a prendere una bacchetta e se la trova occupata rinuncia e riprova più tardi
Problema: STARVATION

Algoritmo 2:
Semaforo per filosofo

Sem_i (0)	Prendi(i)	Posa(i)	Test(i)

Prendi(i)
Sem_i.down()
Posa(i)
Sem_i.up()
	Mutex.down()
	Stato(i) = affamato
	Test(i)
	Mutex.up()
	Sem_i.down()
		Mutex.down()
		Stato[i] = pensa
		Test(i+1)%5 //test al mio vicino destro
		Test(i-1+5)%5 //test al mio vicino sinistro
		Mutex(up)
			If(stato[i] == affamato &&
			Stato[i+1]!=mangia &&
			Stato[i-1]!=mangia)
			Stato[i]=mangia
			Sem_i.up()

Se ci sono le condizioni, ho le risorse disponibili, allora vado avanti, altrimenti mi addormento.

Esempio:

Filosofo 1	Filosofo 2	Filosofo 3

Mangia . . . Posa Pensa Test(f2) //f2 è affamato? Sì, f3 sta ancora mangiando? Sì, allora non sveglio ancora f2 .	Prendi Affamato Sem_2.down() . . . . Mangia Sem_2.up()	Mangia . . . . Posa Pensa Test(f2) //f2 è affamato? Sì, f3 sta ancora mangiando? No, allora sveglio f2 .

Se vogliamo lavorare avremo bisogno di due risorse e se sono a disposizione è solo perché i miei vicini sono in un stato in cui non le stanno utilizzando
Non guarda le singole risorse ma si preoccupa di capire quali sono gli stati dei miei collaboratori per cui io posso proseguire

LEZIONE 32 (lettori-scrittori)

Altro protocollo di sincronizzazione

Rapporti tra processi:

- Lettori: concorrenza
- Scrittori: mutua sclusione
- Lettore-scrittore: mutua escusione

Rispetto al normale Mutex, qui si aggiunge un certo numero di lettori che possono lavorare parallelamente.

Risorse:

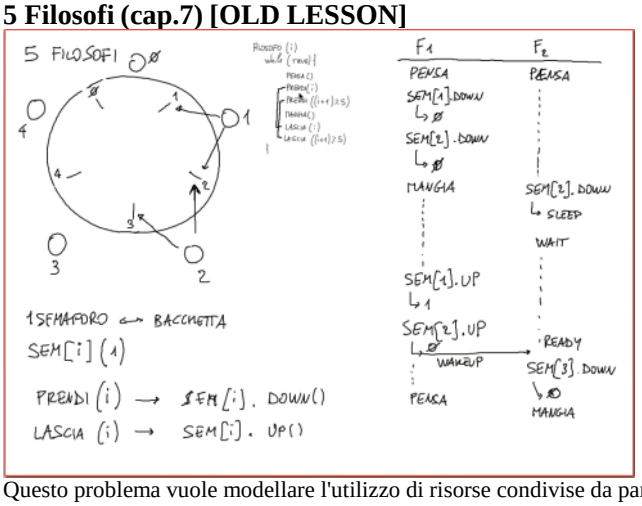
- Mutex(1) //semaforo mutua esclusione
- DB(1) //semaforo che gestisce l'accesso alla variabile condivisa
- int nLettori = 0 //conta quanti lettori sono attualmente presenti nel sistema

Codice:
N.B. Questo protocollo dà la priorità ai lettori

Scrittore	Lettore

DB.Down()
Scritture
DB.Up()
	Mutex.Down()
	nLettori++ //il lettore vuole iniziare la lettura
	if(nLettori == 1) DB.Down() //dobbiamo essere sicuri che non ci siano altri lettori, ci mettiamo in competizione per l'accesso alla sez critica
	Mutex.Up() //necessaria per evitare codice atomico, in questo modo do la possibilità a più lettori di entrare nella loro fase critica
	Lettura //fase critica
	Mutex.Down() //da la possibilità di aggiornre i contatori senza race condition
	nLettori—
	if(nLettori == 0) DB.Up() //Rilascia per eventuali accessi di lettori/scrittori
	Mutex.Up()

5 Filosofi (cap.7) [OLD LESSON]



Questo problema vuole modellare l'utilizzo di risorse condivise da parte di più processi.
Idea di base:

- c'è competizione per l'uso esclusivo di risorse da parte di 5 processi.
- Abbiamo 5 filosofi seduti ad un tavolo
- Per mangiare hanno bisogno di due bacchette, tuttavia queste bacchette sono in condivisione con il vicino, overo una certa bacchetta può essere usata o da un filosofo o dall'altro
- Solo due filosofi (non adiacenti) possono mangiare contemporeanamente

"Vita del filosofo (Versione 1):" --> possibile DEADLOCK (STALLO)

```
Filosofo(i){
        While(true){
                Pensa()
                Predni(i)
                Prendi((i+1)%5)
                Mangia()
                Lascia(i)
                Lascia((i+1)%5)
        }
}
```

Problema: non è detto che la bacchetta di cui abbiamo bisogno sia libera, quindi dobbiamo aspettare
Soluzione:
- modulare l'accesso alla risorsa bacchetta attraverso una mutua esclusione
- Abbiamo visto come possiamo implementare una sezione critica che andremo a sfruttare per ottenere l'accesse alla risorsa in modo esclusivo
- *Idea*: utilizzare i semafori, in particolare: uso un semaforo associato ad ogni bachchetta, quindi lo uso per ottenere il vi alibera ll'uso della bachchetta

"Codice:"
Sem[i](1) // ho un certo numero di semfori con credito 1
Prendi(i) --> sem[i].down()
Lascia(i) --> sem[i].up()

Problema: DEADLOCK (STALLO) = Tutti i filosofi hanno preso la prima bacchetta(es. quella a destra) ma nessuno può prendere la seconda(es. quella a sinistra)
Soluzione:

"Vita del filosofo (Versione 2):" --> FILOSOFO TIMIDO
```
Filosofo(i){
        While(true){
                Pensa()
                Prendi(i)
                Prova( Prendi((i+1)%5) )
                 - Se libera
                        Mangia();
                        Lascia(i);
                        Lascia( (i+1)%5 )
                 - Se non libera
                        Lascia(i)
        }
}
```

Problema: STARVATION = ogni volta che il filosofo tenta di mangiare trova sempre almeno una bacchetta occupata, così non mangia mai
Soluzione: arbitro esterno che gestisce le richieste, tecniche per affrontare il deadlock (+ avanti)

"Vita del filosofo (Versione 3):" --> FILOSOFO MANCINO
```
Filosofo(i){
        While(true){
                Pensa()
                Prendi(Prendi((i+1)%5)) //prende prima la bacchetta sinistra
                Prendi(1) //poi quella destra
                Mangia();
                Lascia( (i+1)%5 )
                Lascia(i);
        }
}
```

Pro:
Si crea una competizione iniziale sulla bacchetta, questo permette ad uno dei due di addormentarsi senza tenersi risorse.
Si spezza il ciclo di attesa in cui tutti aspettano la bacchetta alla propria sinistra --> no deadlock --> no starvation

- SINCRONIZZAZIONE SUI FILOSOFI -
Utilizzo i semafori sui filosofi, cioè do al filosofo la possibilità di lavorare quando ci sono le condizioni, quindi il filosofo dormirà quando entrambe le due bachette non sono disponibili

Semafori:
S[i] --> semaforo del filosofo
Stato[i] --> identifica cosa fa un certo filosofo
- Pensa
- Affamato
- Mangia

Mutex(1) --> utile per evitare di finire in race condition quando interagiamo con la variabile di stato

"Vita del filosofo (Versione 3):" --> FILOSOFO CON SEMAFORO
```
Filosofo(i){
        While(true){
                Pensa()
                Prendi(i) //prendi tutte le risorse che servono al filoso 'i'
                Mangia();
                Lascia(i);  //posa tutte le risorse
        }
}
```

Meccanismo:
- PRENDI(I){
```
                Stato[i] = affamato
                Test(i) //il filosofo 'i' ha diritto di mangiare?
                Mutex.up
                S[i].down
        }
```

- POSA(I){
  Mutex.down
  Stato[i] = pensa
  Test((i+1)%5) //faccio un test sul filosofo alla mia destra
  Test((i-1)%5) //faccio un test sul filosofo alla mia sinistra
  Mutext.up
  }
- TEST(I){
  If(stato[i] == affamato && stato[i+1] != mangia && stato[i-1] != mangia){ //se le bacchette sono libere, allora mangio
  Stato[i] = mangia
  S[i].up
  }
  }

Domande:
E' possibile avere un deadlock? No, non è possibile che tutti siano affamati contemporaneamente

LEZIONE 33 (Barbiere sonnolento)

Barbiere	Produttore

While(true){
cli.down()
mutex.down()
inAttesa—
Mutex.up()
Barb.up()
Taglia()
}
	Mutex.down()
	If(inAttesa < n){
	inAttesa++
	cli.up()
	mutex.up()
	barb.down()
	riceviTaglio()
	}
	Else { mutex.up() }

Caratteristiche:
- se non ci sono clienti, il baribiere dorme
- se no ci sono sedie, il cliente se ne va via
- il cliente si siede e aspetta il suo turno
- il barbiere viene svegliato da ciascun cliente per ottenere il suo servizio

Risorse:
- 1 barbiere
- N sedie
- Cli(0) = semaforo cliente
- Barb(0) = semaforo barbiere
- Mutex(1)
- Int inAttesa = 0 //variabile condivisa che conta quanta gente c'è in coda

LEZIONE 34 (deadlock)

Deadlock: (problema di sincronizzazione) :
- più processi vengono eseguiti in modo concorrente e ci sono più risorse su cui si richiede l'accesso esclusivo
- Esiste un insieme di processi in cui ogni processo attende un evento che un altro processo nell'insieme può generare

esempio: (Filosofi): tutti i filosofi prendono la stessa bacchetta (es. tutti quella alla loro destra)

Processo 1	Processo 2
R1.down R2.down … R2.up R1.up	R1.down R2.down … R2.up R1.up

Condizioni necessarie per avere un deadlock:
- Le risorse sono usate in modo esclusivo, nesusn altro usa la stessa risorsa contemporaneamente, no condivisione
- Possesso e attesa (hold & wait): prendo una risorsa, la acquisisco, poi passo alla successiva; per acquisire la risorsa successiva che mi serve attendo di aver ottenuto quella precedente
- Assenza di prelazione: se ho a possibilità di sottrarre la risorsa a qualcuno non avrei l'attesa, questo non deve essere possibile
- Attesa circolare: p(i) aspetta p(i+1) che aspetta p(i+2) …

Esempi:
- Versione filosofi tutti destri: deadlock (rispetta tutte e 4 le proprietà)
- Filosofo mancino: no deadlock (uno dei filosofi spezza l'attesa circolare)
- Semafori associati ai filosofi: no deadlock (hold&wait non verificata, non prendiamo le bacchette una alla volta)
- (Confronta gli altri esempi visti)

LEZIONE 35 (Grafo di Holt)

Grafo di holt: modo per osservare una fotografia del sistema e capire lo stato delle richieste della allocazioni delle risorse; ci permette di visualizzare la situazione corrente (risorse acquisite e richieste)

- Avremo due tipi di nodi:

Processi	Risorsa

- Avremo due tipi di archi:

Assegnamento	Richiesta

Posso capire da questi grafi se c'è il deadlock?

LEZIONE 36 (gestione deadlock - Banchiere)

Prevenzione del deadlock

- Evitare una delle 4 condizioni necessarie:
  - Richiedere tutte le risorse contemporaneamente (evitare hold&wait)
  - Prevedere la prelazione delle risorse (es. con context switch salvi lo stato)
  - Ordinare le risorse (richiedo le risorse secondo un certo ordine, spezzo l'attesa circolare)
- Evitare il deadlock da parte del s.o.
  - Se conosco le richieste future posso fare dei ragionamenti per gestire al meglio le risorse
- Rilevazione
  - lasciamo che il deadlock avvenga, ma periodicamente facciamo un controllo per vedere se tutto va bene ed in caso allertare o uccidere i processi coinvolti nel deadlock

Esempio:

PA	PB
A.down B.down … B.up A.up	B.down A.down … A.up B.up

Politica di gestione delle risorse:

- richiesta risorse
  - s.o. → ordine assegnamento risorse (sequenza sicura)
    - garantisce assenza deadlock
    - stato sicuro → no deadlock
    - stato non sicuro → deadlock? (non è detto)

Esempio: [Sequenza sicura: P1 → P2 → P0] -- 28 min

	HA	MAX
P0	3	9
P1	2	4
P2	2	7
	3	FREE

	HA	MAX
P0	3	9
P1	4	4
P2	2	7
	1	FREE

	HA	MAX
P0	3	9
P1	-	-
P2	2	7
	5	FREE

	HA	MAX
P0	3	9
P1	-	-
P2	7	7
	0	FREE

	HA	MAX
P0	3	9
P1	2	4
P2	2	7
	3	FREE

	HA	MAX
P0	9	9
P1	-	-
P2	-	-
	1	FREE

	HA	MAX
P0	-	-
P1	-	-
P2	-	-
	10	FREE

Algoritmi per l'analisi della situazione del sistema:

- 1 risorsa per tipologia (1 solo pallino per risorsa)
  - Versione 1 (genera cicli → deadlock):
    - Archi di prenotazione
    - Ipotesi
  - Versione 2 (infattibile):
    - Analisi di presenza di cicli nel grafico
    - Possibile se conosciamo gli archi di prenotazione!!!
- N risorse per itpologia (ALGORITMO DEL BANCHIERE) min. 33
  - …

Risoluzione:

- Elimino 1 o n processi + reset risorse
- Checkpoint → rollback
- Prelazione (intervento complicato)

LEZIONE 37 (gestione della memoria RAM)

Programma + dati → RAM + mem. Secondaria
Rilocazione del codice a tempo dinamico
Fase di passaggio da stato new a stato ready → allocazione memoria + risolvo indirizzi relativi

Indirizzo relativo vs indirizzo dinamico
Sorgente(alto livello) – compilatore(basso livello) – oggetto – linker (ogg, lib, modularità) – eseguibile(runtime) – loader(carica)[assegno indirizzi / visione logica] – immagine in memoria

Esempio (min. 17)
Immagine logica – spazio di indirizzamento parte da 0
Compito s.o.: traduzione tramite MMU da insirizzi logici a indirizzi fisici
Processo genera indirizzi logici

GESTIONE DELLA MEMORIA [OLD LESSON]
Memoria: elenco di byte/word ad accesso diretto e con indirizzamento (possibilità di accedere ad ogni elemento a tempo costante)

Tipologie:
- Memoria principale: ha poca capacità, qui sarà memoriazzato il programma da eseguire e i dati (static o dinamici) predisposti all'avvio del programma o durante la sua exec
- Memoria secondaria: ha più capacita ma è più lenta

Importante: tenere una visione logica della nostra memoria nel SO, perchè altrimenti se andiamo a finire sugli indirizzi della ram non riusciamo più ad avere informazioni sull'organizzazione di questi dati

Scopo:
Meccanismo per far coesistere tutte le informazioni dei vari processi garantendo che questi non si diano fastidio

Funzionamento:
Durante il passaggio da New a Ready c'è una fase in cui il processo viene allcoato in memoria, questa fase è molto influenzata dallo stato del sistema e dalle politiche che si stanno usando per la gestione dei dati. Superata questa fase il processo sarà pronto per essere eseguito. Quindi la responsabilità della gesdtione della memoria è anche quella di prendre il nostro programma eseguibile e associato un pcb, prendere le informazioni scritte ad esempio su un disco (istruzioni che compongono il nostro programma eseguibile) e andarle ad allocare in memoria in modo che questo porti alla definizione in memoria dell'immagine utilizzabil ein esecuzione. Non è detto che questa si auna copia 1:1 (non basta prender eil file eseguibile e copiarlo in memoria, la procedeura è più complessa!)

Problemi:
- Supponeniamo di avere:
  - la ram
  - l'immagine logica di un processo: vede tutte le informazioni relative al processo (codice + dati) in una visione logica, quindi indipendente da una allcoazione fisica (organizziamo i dati partendo dall'indirizzo 0)

Considerazioni:
Nel momento in cui abbiamo una visione logica, tutti iriferimenti ai dati e al codice del nostro programma evono essere considerati con molta attenzione!

Riferimenti a dati o codice:
- Assoluti: se i riferimenti sono fatti in modo assoluto, significa che io vado a scrivere direttamente nel mio codice l'indirizzo definitivo. Quindi un riferimento assoluto rischia di creare un grosso problema, vorrebbe dire che io parlo sempre do un indirizzo ben deciso e non do al SO la possibilità di spostarlo in posizioni diverse.
- Rilocabile: ovvero nel momento in cui il SO decide di allocare il processo in una certa porzione di memoria, devo poi avere la possibilità di aggiornare tutit i miei riferimenti in modo da adatarsi a questo spostamente; quindi, invece di scrivere direttament e un numero prefissato, do la possibiltà al SO di aggiornare queste informazioni
- Relativi: soluzione utile quando si parla di cose vicine (quello che ti interessa te lo esprimo in termini dell amia posizione --> es. salta a 40 byte prima)

BINDING DEGLI INDIRIZZI: ci permette di creare un'associazione corretta tra i riferimeti a livello logico e quelli fisici; ci si occupa di aggiustare gli indirizzi rilocabili in modo che quello che intendevo a livello logico viene preservato a second adella configurazaione finale del mio processo

Esempio (compilazione tipica):
Fasi:
- (fase iniziale) sorgente: file.c, osserviamo come sono gestiti gli indirizzi (simbolici) (es. riferimento a memoria o dati) --> alto livello
- (prima fase)
  - compilatore: ogni file sorgetne viene convertito/tradotto in un file oggetto
    file oggetto: contiene la prima traduzione verso il codice macchina per l'architettura per cui si sta compilando, ma non è ancora eseguibile
  - (dal punto di vista della memoria) l'indirizzo simbolico viene convertito dal compilatore in un indirizzo relativo o rilocabile
- (seconda fase)
  - linker: prende i file oggetto e le librere per creare il file esegubile; sostituisce tutti gli indirizzi simbolici rimasti in sospeso che si riferiscono ad un altro oggetto che il linker può vedere
    file esegubile: contiene la traduzione finale in codice machcina, può essere eseguito
  - (dal punto di vista memoria) si costruisce la memoria complessiva che il programma andrà ad utilizzare unendo i vari pezzi relativi ai vari codici oggetti che abbiamo unito, eventuali riferimenti a file oggetto dovranno essere definiti; riferimenti definiti come indirizzi relativi o rilocabili
- (terza fase)
  - loader: porta tutte queste informazioni in memoria, caricamento effettivo
    Caricamento: gestione di tutti i riferimenti rilocabili, collegamento per decidere qual è la loro destinzaione fisica; carica le librerie dinamiche
    Immagine del processo in memoria: entità dinamica con cui avrà a che fare il SO
  - (dal punto di vista della memoria) la startegia di allocazione della memoria influisce sul caricamento

TECNICHE DI GESTIONE DELLA MEMORIA
Dobbiamo:
- Capire come lavora la MMU: farà un'astrazione della gestione della memoria, non vuole far sapere al programma la posizione fisica dei suoi dati
- (a livello di SO) Gestire tutto a livello software, in modo da poter lavorare correttamente

N.B. Il programma ragionerà sempre in termini di indirizzi logici, non saprà mai l'allocazione fisica associata

Dati:
- Immagine logica del processo, con cui il processo stesso parla (creata e organizzata dal compilatore)
- Convertitore da indirizzi logici a fisici che ci assicura che l'immagine è effetivamente scritta in RAM

Schemi di allocazione della memoria:

- Monotasking -

- Allocazione contigua: se due indirizzi sono consecutivi a livvello logico, questi indiriziiz saranno anche consecutivi a livello fisico (blocco dati mappato come unico blocco indivisibile)
  Meccanismo:
  - Assegno tutta la memoria a un processo (es. MSDOS, il SO coesisteva con il processo) --> soluzione delicata, problematica
  - Partizioni di memoria: la memoria principale viene divisa in 2 (partizione per il SO, partizione per l'utente/processo)
    Come facciamo ad allcoare l'immagine logica del nostro processo in memoria fisica? Non possiamo più far coincidere indirizzi logici e fisici

- Base + Limite: permette di posizionare il nostro processo in una posizione che non è quella iniziale
  Meccanismo:
  - La cpu genera un indirizzo logico "i" che si riferisce allo spazio logico del processo
  - Compito della MMU:
    - Se l'indirizzo logico è sotto il <u>limite</u> (quindi l'indirizzo è valido) --> ok --> sommiamo l'indirizoz logico "I" alla nostra <u>base</u> --> indirizzo fisico (RAM)
    - Altrimenti --> errore

  Vantaggi:
  - Possiamo spostare la base nel tempo, dobbioma poi preocucparci di riaggiornare gli indirizi fisici
  - Possiamo gestire eventuali modifiche al nostro sistema, ad esempio se il SO ha bisogno di allargare la propria memoria, si può:
    - Slittare la base più avanti e ricalcoalre gli indiriziz fisici tramite la MMU --> costoso, prevede tante copie in memoria

- Multitasking -

- Partizioni fisse: divido la memoria in più partizioni fisse, ognuna delle quali occuperà un processo + una per il SO
  Meccanismo:
  - Avrò un numero di partizioni fisso e una dimensione per le partizioni fissa, ogni processo avrà la propria base assegnata al proprio PCB

  Svantaggio:
  - Non posso avere un processo che copre più partizioni (1 processo, 1 base, 1 partizione) --> limite per processi più grandi, spreco per processi più piccoli

  Anche qui si può applicare la tecnica base + limite

- Partizioni variabili: divido la memoria in più partizioni fisse, ognuna delle quali occuperà un processo + una per il SO
  Meccanismo:
  - Il numero di partizioni e la loro dimensione sono decise in modo dinamico
  - Avrò una tabella che mi dice quali partizioni sono libere e quali occupate
  - Allocazione --> trovo partizione libera --> occupo la partizione, aggiorno con nuovi dati processo + aggiorno nella tabella la partizione libera
  - Deallocazione --> libero partizione occupata + unisco partizioni libere consecutive

  Svantaggio:
  - Dopo un po' che vado avanti posso trovarmi con pezzi di memoria libera che non sono più contigui

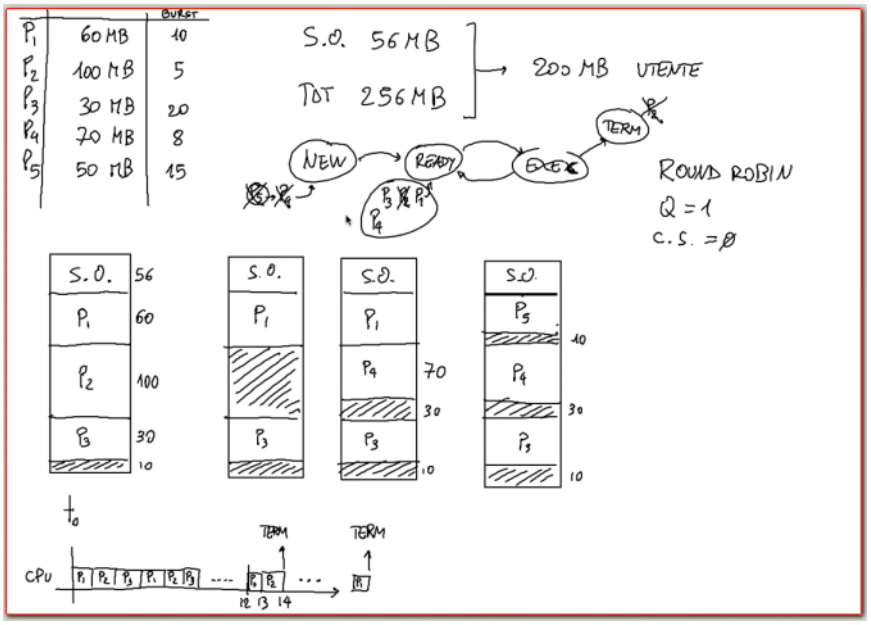
  Come risolvo il problema:
  - Dovrò spostare e rilocare i processi in modo da collezionare tutto lo spazio libero insieme e ottenere così lo psazio libero contigo --> costoso, prevede tante copie

<u>Tecniche di selezione dello spazio libero:</u>

Situazione:
- arriva un nuovo processo, devo gestire la sua richiesta di allocazione --> selezione spazio libero

Tecniche (non più utilizzate):
- First fit: cerco il primo spazio libero sufficientemente grande
- Best fit: cerco lo spazio più piccolo sufficiente al processo
- Worst fit: cerco lo spazio più grande in modo da non lasciare fette minuscole che nessuno userà mai e che mi porteranno a fare uno spostamento per unire le varie parti

Esempio: (lezione 18.2 minuto 40 circa)



Abbiamo capito che l'allocazione dinamica non risolve tutti i problemi:
- Lo spazio libero risulta frammentato
- Altri problemi vari…

LEZIONE 38 (allocazione contigua) --> 12/11/2020

Schemi di allocazione della memoria:

- **Allocazione contigua**

  Se due indirizzi sono consecutivi a livello logico, allora questi indirizzi saranno anche consecutivi a livello fisico.
  L'insieme di dati sarà mappato sul disco come un unico blocco indivisibile.

  Meccanismi:
  - Assegno tutta la memoria ad un processo
  - 2 partizioni di memoria, la prima assegnata al SO mentre la seconda assegnata all'utente per l'esecuzione dei processi
  - Base e limite: tecnica che ci permette di posizionare il processo in una posizione che non è quella iniziale.
    In pratica, la CPU genera un indirizzo logico per il processo e tramite una condizione decide se proseguire o meno;
    la condizione dice: se l'indirizzo logico è minore del mio limite, allora l'indirizzo è valido e possiamo proseguire, altrimenti no e abbiamo un errore.
    Proseguendo abbiamo un'altra informazione, la base, cioè la posizione di inizio memorizzazione fisica (ind. logico + base = ind. fisico)

Partizioni multiple --> multitasking
Abbiamo:
- Numero di partizioni fisso
- Dimensioni di ogni partizione fissa
- Ogni processo è associato ad una propria base

Problema:
- I processi sono limitati dalla dimensione fissa della partizione

Partizioni variabili --> multitasking
Abbiamo:
- Numero di partizioni e dimensioni variabili
- Tabella partizioni libere/occupate
- Allocazione partizioni libere --> dopo l'aggiornamento avrò una partizione occupata e una libera
- Deallocazione partizioni occupate --> dopo l'aggiornamento unisco le parti libere

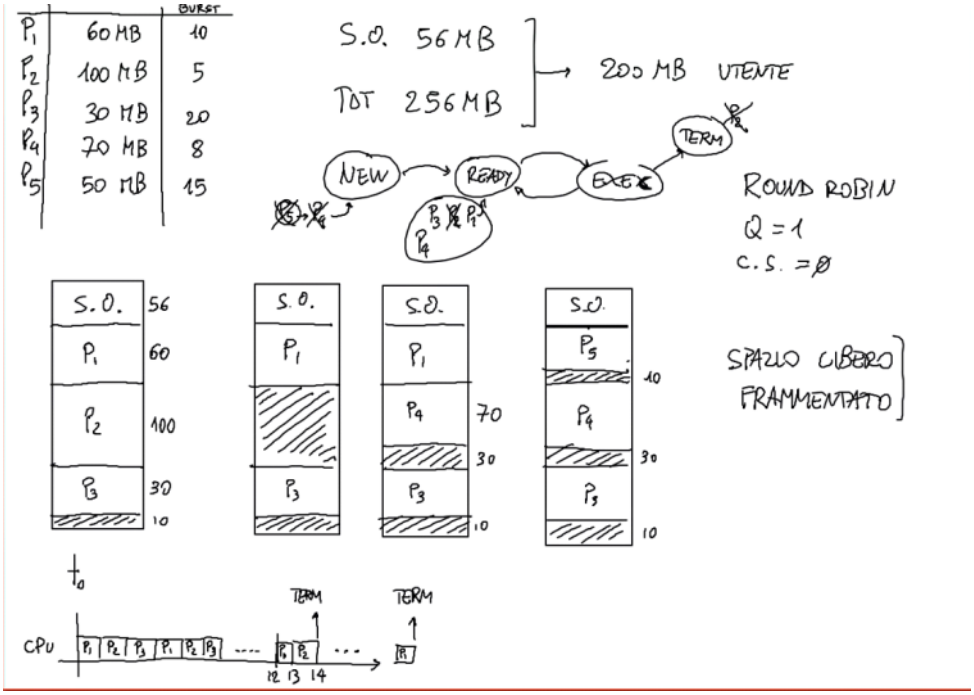
Problema:
- Mi ritrovo ad avere pezzi di memoria libera non contigui

Soluzione:
- Spostare e rilocare i processi in modo da unificare lo spazio libero

Selezione spazio libero
Supponiamo di avere diverse zone libere, quale spazio assegno al mio processo?
- First Fit: occupo il primo spazio libero abbastanza grande
- Best Fit: fra tutti gli spazi abbastanza grandi, occupo lo spazio più piccolo

- **Allocazione dinamica**

  Spiegazione con esempio: lezione "Allocazione contigua" - minuto 38:00



Problema:

- Spazio libero frammentato

LEZIONE 39 (frammentazione) --> 16/11/2020 (pt.1)

- **Esterna**
  Lo spazio totale libero è maggiore di un processo ma non è contiguo

  Soluzione
  Compattazione: sposto/riloco tutti i processi in modo da ottenere un unico spazio libero contiguo
  Problema: attività molto costosa!

- **Interna**
  Spreco di spazio all'interno della partizione fissa

	Frammetazione Interna	Frammentazione Esterna
Allocazione Fissa	SI	SI
Allocazione Variabile	NO	SI

Allocazione non contigua
Lo spazio libero viene occupato anche se la dimensione è minore del processo.
Il processo viene suddiviso in più parti che andranno ad occupare tutte le varie porzioni di spazio libero.

Vantaggi:

- Frammentazione esterna eliminata!
- Non c'è bisogno della compattazione

Osservazioni:

- Devo tenere traccia delle allocazione dei pezzi, quindi avrò una base per ogni parte/blocco di processo (N.B. ogni blocco è contiguo)

LEZIONE 39 (paginazione) --> 16/11/2020 (pt.2)

Paginazione (Paging) [Tipologia di allocazione non contigua]
Lo spazio logico di un processo è diviso in tanti blocchi chiamati PAGINE, tutte della stessa dimensione.
La RAM viene vista come un "contenitore di pagine" e viene divisa anch'essa in tante partizione chiamare FRAME (della stessa dimensione dei blocchi del processo)

Tabella delle pagine
Per ogni pagina ci dice dove è stata memorizzata nella RAM, quindi a quale frame è associata.
Ogni processo ha una propria tabella delle pagine che viene memorizzata nel PCB.
E' rappresentata come un array dove l'indice corrisponde al numero di pagina e il valore è il numero di frame.

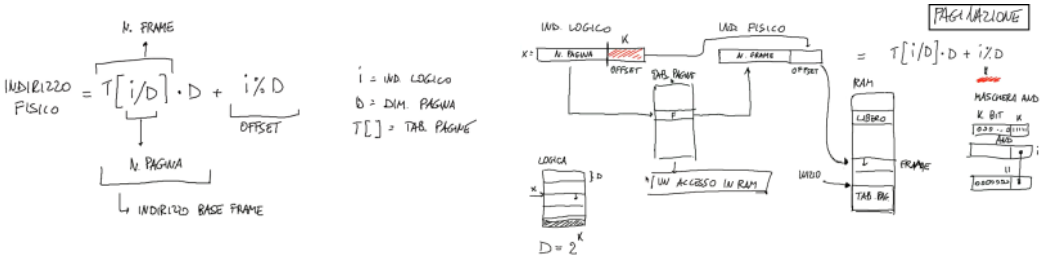
Dimensione della pagina
Solitamente è decisa direttamente dall'architettura.
Considerazioni:

- Dimensione grande: (problema) spazio libero all'interno della pagina --> spreco di frammentazione interna
- Dimensione piccola: (problema) troppe associazioni/puntatori, tabella delle pagine enorme

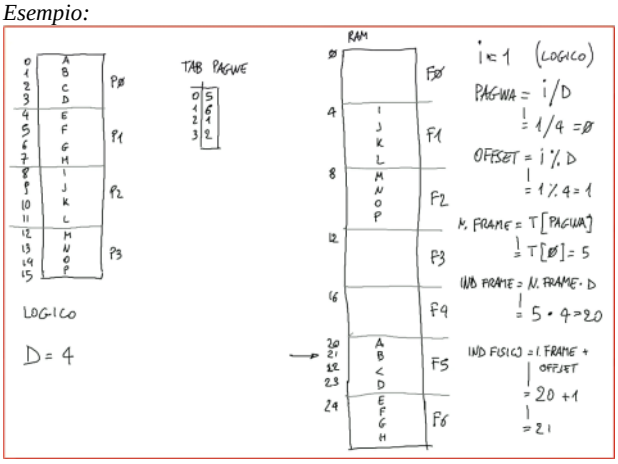
- Dimensione media: ~1KB ottimale

MMU
Supponiamo di voler recuperare un dato all'interno di un pagina di un processo.ss

Offset: distanza/spostamento tra l'inizio della pagina e la posizione del dato che voglio leggere.
Step:
- Identifico la pagina
- Risalgo al frame della pagina tramite la tabella delle pagine
- Identifico l'offset e lo sommo al frame per ottentere il dato:



Esempio:



Osservazione:
- La lettura della tabella delle pagine raddoppia i tempi di accesso in RAM
- Non c'è frammentazione esterna
- C'è frammentazione interna (< D)

Costo:
Accesso alla tabella delle pagine + accesso in memoria

Come gestire e ottimizzare l'accesso in memoria?
Idea 1: tramite dei registri dedicati
Osservazioni:
- Memoria veloce --> efficiente
- Costosa --> insostenibile!

Idea 2: cache + registri
Osservazioni:
- Stesse prestazioni di "idea 1"
- Molti meno registri

Come funziona?
1. ho una tabella chiave - valore (n° pagina - n° frame) che viene chiamata **memoria associativa (TLB: translation look-aside buffer)**
2. Per trovare una certa pagina devo fare una **consultazione** della memoria associativa, cioè un controllo di tutte le chiavi presenti nella tabella:
   - Se trovo la chiave = ho un **HIT**, ottengo il numero di frame
   - Se non trovo la chiave = ho un **MISS** --> devo cercare il frame nella tabella delle pagine
3. Aggiorno la cache:
   - Se una cosa mi è appena servita, allora probabilmente mi riservirà a breve
   - LRU (Last recently used) --> butta via la pagina usata meno recentemente

Costo:
Accesso alla TLB + accesso in memoria

Esempio:
Dati:
- Hit Ratio = 0.80
- Hit Time = 20 nS
- Hit Penalty = t.accesso ram = 100 nS

Risultato:
- Tempo medio per accesso

= Hit Time + (1 - Hit Ratio) * Hit Penalty + t.accesso ram
= 20 + 0.2*100 + 100 = 140 nS

LEZIONE 40 (paginazione a più livelli) --> 18/11/2020

Tabella delle pagine (altre informazioni)
E' possibile aggiungere altre informazioni alla tabella delle pagine, come ad esempio:
- (bit) Read Only: la pagina è in sola letutra --> genera protezione e v ain eccezione se si tenta una scrittura
- (bit) Validità: 1 = frame valido / 0 = frame non fa parte del processo

Problema:
- La tabella delle pagine è contigua --> frammentazione!!!

Soluzione 1:
Osservazioni:
- Quanto è grande la tabella?
  Dati:
  - 8GB ram
  - 1KB dim.pagina (2^10B --> 10bit offset)
  - 32bit n.frame (int)
  Risultato:
  - Dim. Tab. pagine = 8M * 32b = 8M * 4B = 32MB
  Nota bene:
  - Evidentemente la tabella non può stare dentro un frame (che è grande solo 1KB)
- Quanti frame occupa la tabella?
  = 32MB/1KB = 32K frame --> devono essere bloccati consecutivamente!

Soluzione 2:
Utilizzo la paginazione, *come?*
- Divido la tabella delle pagine in pezzi i quali vengono anch'essi associati ai vari frame della ram.

Osservazioni:
- Così facendo genero una nuova tabella per poter gestire i puntatori/associazioni, chiamata **tabella delle tabelle delle pagine**
- Nel PCB verrà memorizzata la radice di questo albero conenente tutte le tabella delle associazioni

Domanda:
Siamo sicuri che la tab. delle tab. delle pagine sia abbastanza piccola da rientrare in un frame?
- Se è abbastanza piccola --> ok
- Altrimenti aggiungiamo un altro livello di ricorsione, quindi spezzettiamo anche questa tab. e creiamo una nuova tab. per i puntatori di questa tab.

Condivisione delle pagine
Scopo: Area di memoria condivisa tra due processi
Come: assegno/punto ai vari indirizzi logici delle pagine dei vari processi lo stesso indirizzo fisico del frame della ram.

LEZIONE 41 (segmentazione) --> 23/11/2020 (parte 1)

Segmentazione
Suddiviso il mio progremma in tante parti/segmenti, come ad esempio:
- Codice
- Dati
- Stack

Ogni **segmento** è un'area contigua di memoria la cui dimensione è stabilita in base alle necessità.
Visione logica: insieme di segmenti
Indirizzo logico: <n.segmento, offset>

Conversione logico --> fisico
Offset < Limite?
- Sì: base + offset = indirizzo
- No: segmentation fault!

HW --> **Registri**
- Associativi: ottengo informazioni della tabella
- Di segmento: base corrente (codice, dati, stack)

Osservazioni:
- Può dare **frammentrazione esterna**, non so più dove andare ad allocare le informazioni (partizioni variabili)
- Non può esserci frammetnazione interna perché le partizioni vengono create apposta della giusta dimensione

Perché nella paginazione non c'è frammentazione esterna?
Il fatto di avere pagine di dimensioni tutte uguali ci permette di scambiarle in ram dove c'è posto, mentre con dimensioni varoabili non è possibile.

Segmentazione paginata
Obiettivo:
- Togliere la frammentazione
- Gestione del codice rilocabile con segmentazione

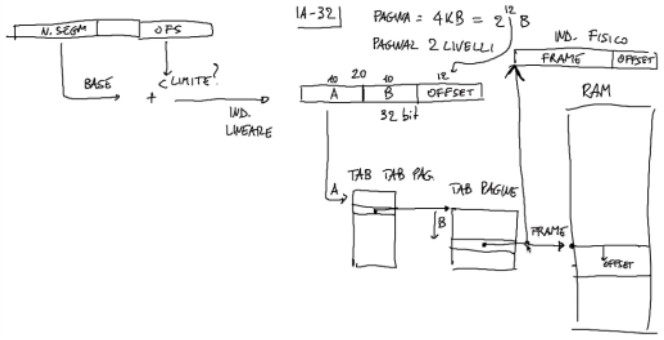
Intel IA-32
CPU ---(ind.logico)--> SEGMENTAZIONE ---(ind.lineare)--> PAGINAZIONE ---(ind.fisico)--> RAM

Osservazioni:

- Segmento limite massimo = 2^32 = 4GB (possiamo indirizzare al massimo 4GB di informazioni)
- Processo massimo = 16K segmenti
  - 8K segmenti processo
  - 8K segmenti condivisi
- Tabelle
  - Tabella locale (descrittori segmento) LDT
  - Tabella globale (descrittori condivisi) GDT
  - Ogni record contiene 8Byte (contiene info sulla base, limite, …)
- Indirizzo logico = <selettore_segmento, offset>
  - Selettore_segmento = 16bit --> "^16 = 8K
    - 13bit = numero segmento
    - 1bit = LDT/GDT
    - 2bit = protezione

Registri di segmento (messi a disposizione dalla macchina):
- CS (Code)
- DS (Data)
- SS (Stack)
- ES (Extra)
- …

Esempio: (simile a pagina 346 del libro Silberschatz)



LEZIONE 41 (memoria virtuale) --> 23/11/2020 (parte 2)

Vantaggi:
- Eseguire un processo non completamente in ram
- Eseguire un processo più grande della ram
- Visione logica della memoria della macchina
- Condivisione della memoria
- Prestazioni ottimizzate

Oservazioni:
- Immagine di un processo *(simile a pagina 355 del libro Silberschatz)*
- Le pagine che "servono" sono molte meno della dimensione del processo

Obiettivo:
Abbiamo bisogno di memoria, più esattamente:
- RAM: più veloce, ma più piccola
- Memoria secondaria: più lenta, ma più grande

Cerchiamo di prendere il meglio di entrambe le parti e formiamo la **memoria virtuale**, cioè *velocità + capacità.*

Come?
Sfruttiamo la paginazione:
- Aggiungiamo alla pagina logica un bit che ci dice se la pagina si trova in RAM o in Mem.Secondaria
- Aggiungiamo la possibilità di spostare una pagina dalla RAM alla Mem.Secondaria e vicecersa:
  - SWAP IN: carico pagina in RAM
  - SWAP OUT: sposto pagina su Mem.Secondatia

SWAPPER (pigro / lazy)
Scopo:
- Minimizza il traffico, viene usato solo quando strettamente necessario (il processo chiede una pagina)

Meccanismo:
- Processo in exec
- Genera l'indirizzo alla MMU, ma la MMU si accorge che la pagina non è presente
- **PAGE FAULT**: eccezione per pagina non presente in RAM
- Il sistema operativo:
  - sposta il processo nello stato di wait
  - Identifica un frame vuoto
  - Copia dal disco alla ram
  - Termina copia
  - Aggiorna la tabella delle pagine
  - Aggiorno bit + frame
  - Processo in stato ready
  - (ripetere ultima istruzione)

Potrebbe capitare che il processo parte senza pagine in RAM, allora lo swapper / page fault caricano le pagine che servono.

Nota:
Così facendo non avremmo un continuo swap in / swap out e di conseguenza un costo elevato? No, perché:
Principio di località: probabilità alta di usare ancora le pagine appena usate (es. cicli)
Quindi, le pagine che mi servono sono caricate dallo swapper e chiedo pochi page fault; il tempo di accesso in memoria è simile al tempo di accesso alla ram

Riavvio istruzione dopo page fault
Fasi:
- IF
  - (problema) non trovo l'area di memoria che contiene l'istruzione da eseguire
  - (soluzione) carico la pagina che contien eil codice e riparto
- ID
- EX
- RAM
  - (problema) non trovo il dato in input, l'allocazione per scrivere l'output
  - (soluzione) rieseguo l'istruzione, quindi rifaccio il fetch + istruzioni
- WB

Esempio:
CISC ---> MOUSB: copie di array in array
Cosa succede se scopro che una certa cella appartiene ad una pagina che non è presente in RAM?
Devo annullare tutta la singola istruzione e successivamente dovò riavviarla (causa molti problemi a livello di cpu)

Tempi di accesso = (1 - f) * t.accesso ram + f * t.accesso disco
- f = frequenza di page fault (circa 0) --> più è bassa più il t.a. è basso

LEZIONE 42 (sostituzione pagina) --> 30/11/2020

Il meccanismo di Swapper richiede del tempo per poter recuperare la pagina dal disco, come possiamo velocizzarlo?

METODO 1:
FORK: chiamata al s.o. che duplica il processo corrente

Osservazioni:
- Copio anche le pagine in memoria, quindi la tabella delle pagine punterà agli stessi frame in memoria
- Aggiungo alla tabella il flag COPY ON WRITE: creo una nuova copia di puntatori a nuovi frame della ram solo se necessario quando avviene una modifica

SOSTITUZIONE PAGINE (pagina 367 sul libro)
Premessa:
- La multiprogrammazione mi permette di avere tanti processi allocati in RAM

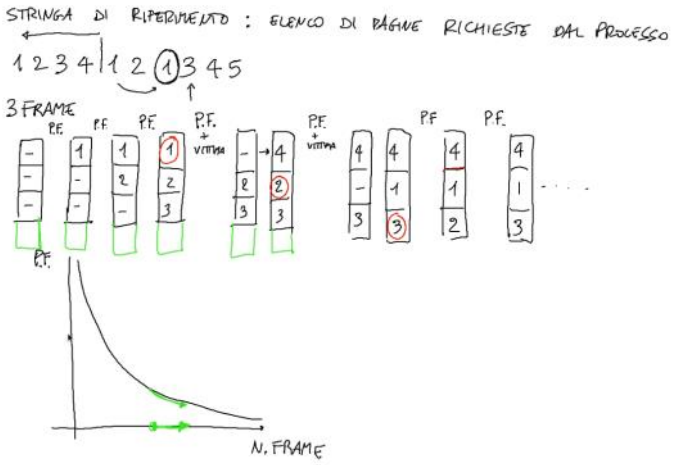
Osservazioni:
- Supponiamo di avere la memoria completamente piena
- un nuovo page fault non può memorizzare un nuovo frame
- Algoritmo di sostituzione pagina: seleziono il frame "vittima" che andrà sostituito
- Swap out "vittima" e swap in "nuova pagina" + aggiorno tab.pagine

DIRTY BIT: indica se c'è stata una scrittura sulla pagina
- 1: quando mando via in swap out la mia pagina, allora delvo fare lo swap out;
  Perché effettivamente significa che la copia che avevo sul disco non è più allineata
- 0: ho un'altra vittima con dirty bit a 0, significa che cul disco ho una copia identica che in RAM non è mai stata modificata, allora non c'è bisogno di fare lo swap out, cambierò solo la tabella delle pagine

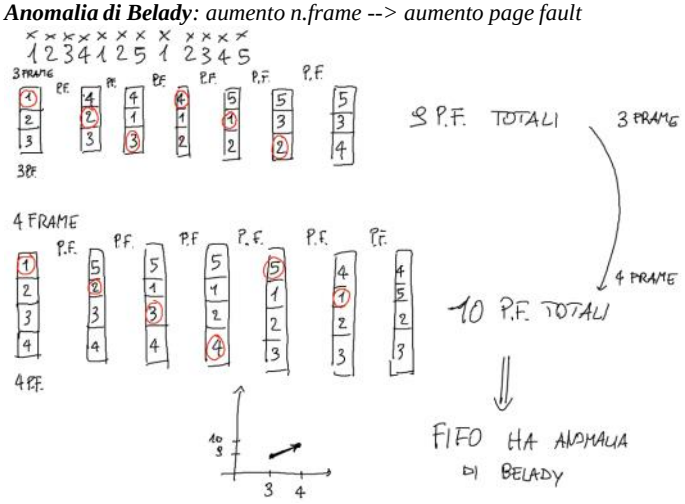
Ho dimezzato i costi, ma bisogno trovare un criterio con cui mandare via le giuste pagine.

Osservazioni:
- Da cosa sono causati i page fault? Come possiamo minimizzarli?
  - Algoritmo di selezione della vittima (se scelgo male potrei essere costretto a ricaricare le stesse pagine poco dopo)
  - Allocazione frame, ogni processo potrebbe essere gestito con un certo numero di frame in RAM (se do tanti frame potrebbe caricare tante info e farà meno page fault e viceversa)
- Quale frame scelgo di mandare via?
  Strategia: prevedo il futuro (principio di località)
  - **Sostituzione FIFO**: la prima pagina che entra in RAM è la prima ad uscire (tempo implicito)

    Esempio 1:



Esempio 2:
***Anomalia di Belady**: aumento n.frame --> aumento page fault*

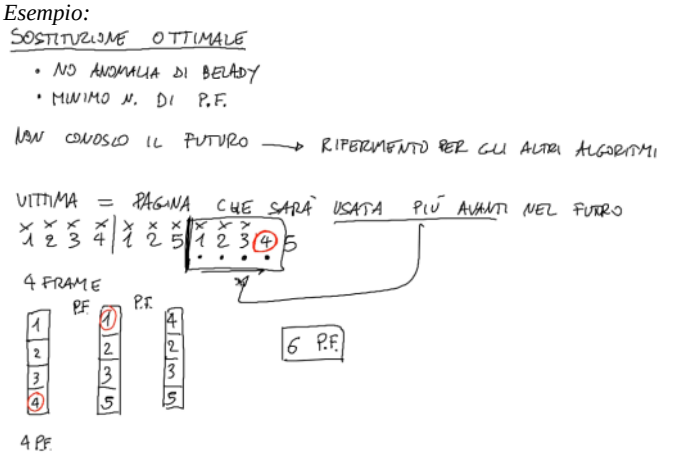


- **Sostituzione OTTIMALE**:
  Caratteristiche:
  - No anomalia di Belady
  - Minimo numero di paga fault

  N.B. Non conosco il futuro --> non è un algoritmo possibile, ma serve come riferimento per gli altri algoritmi!

  <u>Vittima</u> = pagina che sarà usata più avanti nel futuro

  Esempio:



- **Sostituzione LRU (Least Recently Used)**
  Approssimo il futuro vedendolo simile al passato

  <u>Vittima</u> = pagina usata meno recentemente

  Problema 1: ho bisogno di tenere traccia del tempo di accesso di ogni pagina, come?
  - Timestamp: ogni volta che accedo alla pagina aggiorno il timestamp (costo: O(n))
  - Stack: ordino dalla pagina usata meno recentemente a quella usata più di recente (costo: O(n), ma evita i timestamp)

  Problema 2:
  - Permette il riuso delle pagine

Costo: O(n) --> costoso!
Approssimazione (riduco complessità) --> costo: O(1)
Idea:
Invece di usare un timestamp per ogni pagina e poi trovare il minimo, faccio una versione semplificata, come?
Utilizzo un solo bit:
- 0: pagina non usata di recente
- 1: pagina usata di recente

Meccanismo:
- Periodicamente, resetto tutti i bit = 0
- Ad ogni accesso, scrivo il bit di riferimento = 1 (pagina utilizzata)

- **Algoritmo dell'orologio (seconda possibilità)**
FIFO (coda circolare) + bit riferimento

Il puntatore indica la vittima:
- Se il bit di riferimento è 0, allora ho trovato la vittima
  - Carico la nuova pagina
  - Setto il bit di riferimento a 1
  - Avanzo il puntatore
- Se il bit di riferimento è 1, allora setto il bit di riferimento di quella pagina a 0 (reset della pagina)
  - Avanzo il puntatore

LEZIONE 43 (allocazione frame) --> 03/12/2020

Secondo aspetto per gestire al meglio la memoria virtuale.
Quanti frame allocare?
MAX = n.frame disponibili ram (meno pagine condivise)
MIN = dipende dalle istruzioni (e dall'hardware)

Allocazione omogenea
Supponiamo di avere:
- M frame
- N processi

Allora:
- p_i = m/n (tutti i processi hanno gli stessi frame in memoria)

Allocazione proporzionale
Se p_i occupa S_i (dimensione immagine processo)
Allora:
- p_i = S_i / sommatoria(S) * m --> (se > MIN)

Osservazione:
- Non c'è relazione tra la dimensione dell'immagine del processo e il numero di frame allocati in ram

Scopo:
- Il numero di frame da allocare dipendono dal comportamento, e quindi dall'algoritmo applicato

Algoritmo selezione vittima
- Sostituzione globale: vittima scelta fra tutti i frame, anche quelli degli altri processi
  Oss: il processo non può controllare i suoi page fault (non può difendersi dagli altri processi);
  Idea: utilizzare la priorità per controllare hci può rubare (es. un porcesso con priorità più altra può rubare ad uno con priorità più bassa)

- Sostituzione locale: vittima scelta fra i frame del processo (n.frame allocati costante)
  Oss: un processo potrebbe soffrire moltissimo perché ha pohci frame oppure ne ha in abbondanza rispetto a quelli che gli servono
  Idea: il s.o. periodicamente deve aggiornare il numero di frame allocati

Problemi:
- Un processo continua a fare page fault. Cause?
  - Numero di frame insufficiente
  - Algoritmo sostituzione pagina lavora male

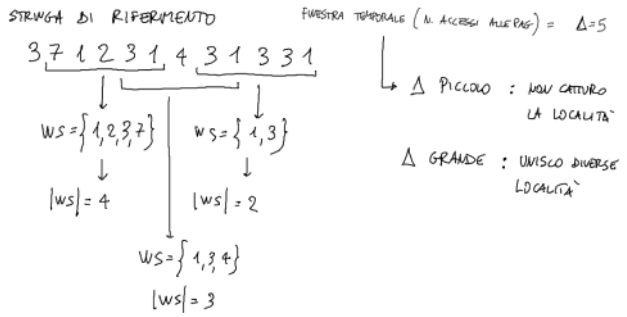
  Soluzione: aumento i frame

- Tutti i processi fanno page fault (sovra allocazione, la ram è limitata!)
  - Tutti i processi hanno necessariamente bisogno di più frame di quelli allocati
  - Il disco è carico e viene riempito di richieste
  - I processi finiscono continuamente in coda Wait --> la cpu non lavora!

  Soluzione: **THRASHING,** diminuire la multiprogrammazione
  - sospendo un processo --> uso i suoi frame
  - Uccido un processo

WORKING SET: insieme delle pagine del processo utilizzate attivamente in un certo periodo
Identifico delle **località**, cioè periodi di tempo in cui le pagine attive sono più o meno costanti.
Durante il periodo di tempo, se il working set è attivo, non ho page fault.

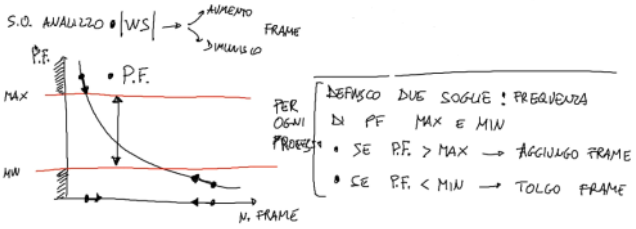
Esempio:



Dimensione w.s. = misura delle richieste alla ram soddisfatte senza page fault; numero di pagine che dovrebbero essere allcoate in ram (<= numero di frame allocati)

Problema:

- Come modificare dinamicamente il numero di frame, al fine di aggiustare l'utilizzo in ram di un processo, in base al suo comportamento?



LEZIONE 44 (gestione periferiche e scheduling disco) --> 09/12/2020

Le periferiche possono avere:

- Tipo diverso
- Velocità diverse

Un sottosistema del kernel si dedica alla gestione di queste diverse periferiche.

Meccanismi per la comunicazione con la periferica:
Il processore può:

- Tramite istruzioni in/out, trascerisce dati sulla porta
- La periferica di I/O viene mappata in ram, e le operazioni vengono effettuate sulla ram, la quale trasferisce poi le richieste al dispositivo

Protocolli di comunicazione:

- Handshaking: il processore dialoga con il controller attraverso una serie di bit
- Interrupt: quando arriva un interrupt si risveglia una parte di kernel che avrà cura di gestire i dati
- DMA

Scopo: avere poche interfacce I/O che si adattano ad ogni dispositivo

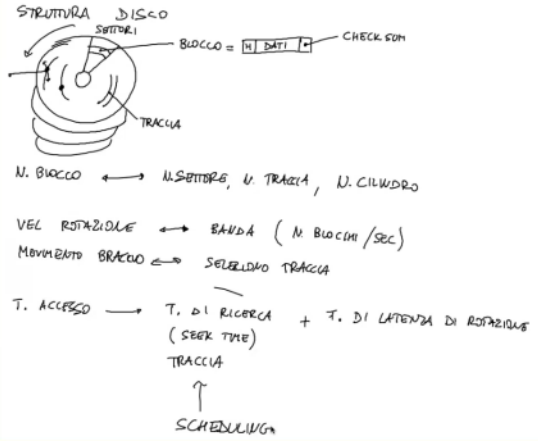
Caratteristiche:

- Trasferimento dei dati:
  - A carattere (stream)
  - A blocchi
- Accesso
  - Sequenziale (tempo di accesso non costante)
  - Diretto
- Trasferimento
  - Sincrono (dobbiamo aspettare in coda wait in attesa)
  - Asincrono (possiamo fare altro e avere un segnale tipo interrupt che ci segnala il completamento)
- Condivisione
  - Mutex
  - Multi utente
- Velocità / tempo di accesso

Servizi (offerti dal sistema operativo per ottimizzare e migliorare l'uso del sistema)

- Scheduling: ordino le richieste
  - Ottimizzo il tempo di attesa
- Buffering: memoria temporanea (non accessibile all'utente, del s.o.)
  - Compensa velocità diverse
  - Adatto dimensioni blocchi diverse
  - Copia delle informazioni da scambiare per garantire all'utente la semantica della copia
- Caching: altro tipo di buffer che mantiene le informazioni più usate
  - Evito di consultare continuamente il dispositivo per informazioni che uso più volte
  - (opt.) Utilizzo di tecniche LRU per mantenere l'informazioni più recente
- Spooling: altro tipo di buffer che accoda le richieste
  - Rende semplice l'accesso alle periferiche che sono usate in mutex e in modo sequenziale accodando le richieste
- Errori: segnalazione / codifica degli errori

Struttura disco



Algoritmi di scheduling disco
Supponiamo di avere una coda di richieste, dove tutti i processi chiedono di leggere/scirvere certi blocchi del disco.

- FIFO
  Serviamo le richieste nello stesso ordine con cui sono arrivate.
  Caratteristiche:
  - Percorso lungo
  - Tempi di servizio alti
  - Tante inversioni

  Conclusione:
  - Non ottimale

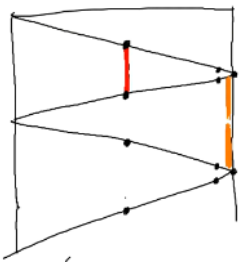
- Shortest Seek Time First (SSTF)
  Serviamo per prima la richiesta che richiede la posizione della traccia più vicina alla testina in questo momento.
  Caratteristiche:
  - Percorsi migliori (meno strada, meno inversioni) rispetto FIFO
  - Ottimizzo localmente, non globalmente
  - Rischio di starvation (se arrivan tante richieste vicine tra loro, quella più distante potrebbe non essere mai eseguita) --> tempi di servizio poco stabili

  Conclusione:
  - Non ottimale

- SCAN
  Parte dalla traccia 0 ed iniza a servire le richieste da lì in avanti, fino a quando arriva all'ultima traccia, allora torna indietro e serve tutte le richieste fino a che ritorna alla traccia 0. (ex. ascensore)
  Caratteristiche:
  - Percorsi maggiori rispetto a SSTF
  - Movimento regolare della testina dalla traccia minima alla traccia massima
  - Se ci sono delle richieste durante la strada, le soddisfa
  - Non c'è starvation
  - Tempi di servizio?

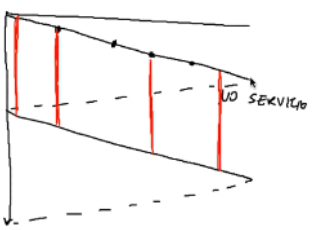
LEZIONE 45 (scheduling disco) --> 10/12/2020 (parte 1)

Analizziamo l'algoritmo di sheduling SCAN.
Tempo di servizo:
- Nel mezzo sono servito con un tempo di attesa dimezzato rispetto agli estremi del disco



Scopo:
- voglio offrire un tempo di servizio che sia uniforme per tutti
- minimizza lo spostamento della testina

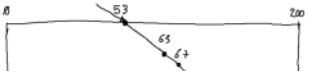
C-SCAN (Circular Scan) --> evoluzione di SCAN
Funziona come l'algoritmo SCAN, con la differenza che durante il ritorno alla traccia 0 non vengono servite le richieste.



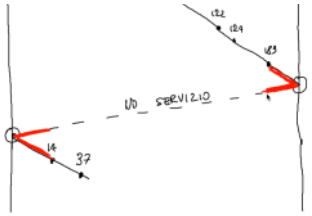
Problema:
- Percorriamo comunque tanta strada che potrebbe anceh essere evitata se non ci sono richieste da servire in quel tratto, come ad esempio il tratto vicino la traccia massima/minima

Scopo: limitare lo spostamento solo se necessario.

LOOK --> evoluzione di C-SCAN
Funziona come C-SCAN, ma evito di arrivare al bordo (mi fermo all'estremo delle richieste)



Caratteristiche:
- All'andata, dopo ogni richiesta mi chiedo se c'è una richiesta più grande:
  - Sì, allora mi sposto avanti e servo la richiesta
  - No, allora torno indietro senza arrivare alla traccia massima
- Al ritorno, dopo ogni richiesta mi chiedo se c'è una richiesta più piccola:
  - Sì, allora mi sposto indietro e servo la richiesta
  - No, allora vado avanti senza arrivare alla traccia 0



Altre caratteristiche da tenere in considerazione nello scheduling:
- Priorità: alcune richieste sono fuori dall'ordine ottimale, non schedulabili = FIFO
- Ordine scrittura: File system + giornale (log delle attività) = Affidabilità
  Es. (ad ogni punto faccio una scrittura sul giornale)
  - comincio a modificare
  - faccio le modifiche
  - alla fine delle modifiche mi segno sul giornale che ho finito di modificare

Amministrazione disco
- Configurazione
  - *Formattazione a basso livello*
    Posso definire la dimensione di settori, tracce e blocchi, di cui definisco:
    - header = #settore, #traccia, codice inizio
    - Dati
    - Terminatore = checksum (controllo correttezza info)
      - C'è satto un errore?
      - Ci sono satti più errori?
      - Correggo errori?

    Come?
    - Bit parità (XOR tutti i dati) --> mi dice se c'è stato un numero dispari di errori
    - Codice di correzione degli errori --> rilevano 2k errori e correggono k errori

  - *Formattazione ad alto livello*

LEZIONE 45 (raid) --> 10/12/2020 (parte 2)

Organizzazione dischi
Insieme di dischi, *osservazioni:*
- Più capacità = maggiori prestazioni se suddivido i dati tra i vari dischi (**Striping**)
- Poca affidabilità (è molto più probabile che uno dei dischi si rompa)

Idea: risolvere i problemi di affidabilità utilizzando la **ridondanza**.

Problema, tipi di guasto:
- Probabilità dipendenti: tutti i dischi si rompono contemporaneamete
- Probabilità indipendenti: la rottura dei dischi (es. D1 e D2) avviene in modo indipendete l'uno dall'altro
  - Tempo medio di guasto = (T_D1 * T_D2) / 2*T.Riparazione
  - Tempo medio di riparazione = 24h

Soluzione: uno dei due dischi contiene una copia identica dell'altro

RAID (Redoundant Array of Independent Discs)
Idea: organizzare i dischi al fine di bilinciare capacità e affidabilità.
Come? Tramite dei livelli, ognuno dei quali identifica certe configuarazioni.

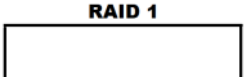
Livelli:
- Livello 0: striping a livello di blocco (massime *prestazioni*)
  Osservazioni:
  - Prestazioni massime
  - No affidabilità
  - Blocco logico i:
    - i % n = in quale disco avviene la scrittura
    - i / n = in qule blocco del disco avviene la scrittura

- Livello 1: mirroring (*affidabilità*, organizzazione per massima ridondanza)
  Osservazioni:
  - Necessario numero pari di dischi
  - Disco amico (i+n) = copia di i

- Livello 4: dedico un disco dedicato a memorizzare la parità (striping + disco parità)

- Livello 5: la parità viene distribuita su tutti i dischi presenti; ogni disco ospiterà dei pezzi di parità (striping + parità distribuita su tutti i dischi) --> prestazioni elevate + un po' di ridondanza

- Livello 0+1:
  Osservazioni:
  - Ho una serie di dischi (striping) = raid 0



□ Ho una serie di dischi di backup (mirror) = raid 1

- Livello 1+0:
Osservazioni:
□ Ogni disco ha una copia di backup (mirror) = raid 1
□ L'insieme di tutte le coppie forma la serie di dischi (striping) = raid 0

LEZIONE 46 (file system) --> 14/12/2020

File system: permette la gestione di tutta l'informazione a memorizzazione permanente.
E' caratterizzato da:

- File: visione logica di una certa informazione
  Contiene:
  - Dati: sequenza di byte
  - Metadati: serie di informazioni che dipendono dal file system (nome, id posizione, protezione, data creazione, utente, …)

  Operazioni:
  - Ricerca nel file system
  - Creazione/apertura
    - Trova spazio libero
  - Lettura/scrittura
  - Cancellazione, spostamento

  S.O. offre:
  - Caching: tabella dei file aperti

  Metodi accesso:
  - Sequenziale (puntatore posizione corrente)
  - Diretto (dipende dal dispositivo, es. nastro)

- Directory: insieme di file/directory (file speciale)
  Contiene:
  - Tabella che contiene un elenco esplicito di file (stringa nome file + riferimento al blocco che contiene il file)

  Operazioni:
  - Ricerca file tramite "nome_file.estensione"

- Partizioni

- Insieme delle operazioni sulle strutture dati che possono supportare (gestione, modifica, …)

Altre considerazioni:
- ??? ~25min lezione file system 14/12/2020
- Montaggio del file system (mount)
- Condivisone (aggiungere al file informazioni per garentire permessi a utenti/gruppi/altri)
- Protezione (lettura/scrittura/esecuzione)

Osservazioni, il f.s. deve essere/avere:
- Permanente
- Algoritmi efficienti --> strutture dati
- Interfaccia (con operazioni ad alto livello)

Implementazione (livelli interni):
1) File System Logico
   - Gestione metadati (no contenuto!)
   - Struttura delle directory (gerarchia, relazioni, …)
   - Descrittore (File Counter Block): Per ogni file c'è un record che contiene tutte le informazioni

   Operazioni:
   - Cerca file
   - Crea/cancella file
   - …

2) Modulo organizzazione file
   Accesso al blocco fisico tramite il blocco logico

3) File System di base
   Legge e scrive i singoli blocchi

4) Controllo I/O
   Driver del dispositivo: si occupa di tradurre le richieste di lettura del blocco fisico e mandare i comandi all'attuatore

LEZIONE 47 (allocazione contigua, collegata, FAT) --> 15/12/2020

Struttura File System:

- Blocco di controllo per il boot: caratteristiche del s.o.
- Partizioni (possono essere viste come dischi separati)
    - Blocco di controllo della partizione (il primo conterrà anche una tabella di tutte le partizioni, descrizione, numero blocchi , numero tracce)
    - Directory
    - Dati

In memoria RAM:

- Gestione
    - Tabella partizioni --> sono montate o no
    - Tabella file aperti
- Cache
    - Copia dei descrittori di file recenti

Apertura file:

- Ricerca file nelle directory (matching stringa)
- Recupero del FCB(File Control Block) identificato e copia nella tabella dei file aperti
- Viene creato un puntatore alla tabella dei file aperti riferito alla tabella del processo corrente
  + posizione di lettura corrente
  + modalità di accesso
- (Linux) viene creato un file "descriptor", cioè un intero che fa riferimento alla tabella
- (Window) viene creato un file "handle", cioè un intero hash che identifica il file

Metodi di allocazione

Premessa:

File (visto logicamente): array di byte/blocchi consecutivi
Attività del sistema:
- Identificare il blocco corretto in cui c'è il primo byte del file

Allocazione contigua

I blocchi logici vengono mappati in modo contiguo anche sul disco.
Directory: nome file, blocco inizio, blocco finale / lunghezza
Vantaggi:
- Movimento minimo della testina
- Tempi di accesso ottimali
- Massima efficienza

Problemi:
- Frammentazione esterna: spazio libero non contiguo!
  Soluzione 1: compattazione? --> No! Troppo pesante.
  Soluzione 2: utilizzo di cd, dvd, nastri dove non è prevista la variazione dei file
- Frammentazione interna: il file ha una dimensione minima di allocazione (cioè la dimensione di un blocco del disco), se ne viene utilizzato meno spazio avremo comuqneu allocato tutto quel blocco --> in base al file system c'è un costo minimo

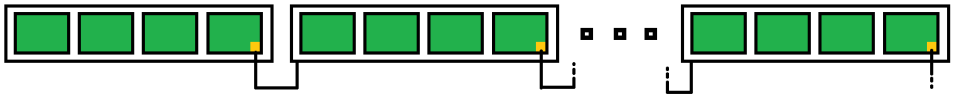
Allocazione non contigua

- Allocazione collegata (Linked List)
  Per ogni blocco, utilizzo alcuni byte per creare un puntatore al blocco successivo; l'ultimo blocco avrà un terminatore che identificherà appunto l'ultimo blocco.
  Directory: nome file, puntatore al primo blocco
  Attività:
    - Scrittura: cerco un blocco libero, lo aggiungo alla lista (aggiorno i puntatori)
  Vantaggi:
    - Non c'è frammentazione esterna
    - Se il file cambia nel tempo, allora allungherò o accorcerò la lista (costo minimo)
    - Non c'è bisogno di compattazione
    - Deframmentazione: utile per avvicinare i blocchi per aumentare le prestazioni
  Problemi:
    - Accesso al file sequenziale (devo leggere un blocco dopo l'altro)

Accesso ai blocchi (liste)

Problemi:
- Accesso sequenziale
- Costo dei puntatori (circa 1% spazio)

Soluzione:
- **Cluster**: se ho una serie di blocchi posso fare un cluster, cioè un'unione logica di k blocchi (k potenza di 2). Considero il cluster come elemento minimo di memorizzazione, così facendo ingrandisco lo spazio e diminuisco di k volte i puntatori.
  Problema: si crea frammetnazione interna
  Soluzione: trovare una dimensione ideale per bilanciare il costo dei puntatori e lo spazio che vado a sprecare per allcoare i file piccoli (es. solitamente è di qualche KB)



Affidabilità

Problema:
- Cosa succede se si rompe un blocco nel disco? In particolare byte del blocco che contiene il puntatore al prossimo cluster?

Soluzione:

- Tengo tutti i puntatori in un'altra struttura separata (tabella di adiacenza, contenente: id cluster, (next)puntatore prossimo cluster) <-- **File Allocation Table (FAT)**
  L'FCB conterrà il riferimento all'entry point della lista, anziché puntare direttamente al blocco.
  Osservazioni:
  - Affidabilità: per garantirla posso usere la tecnica del mirroring (raid 1)
  - Accesso sequenziale: posso andare direttamente in un blocco senza dover scorrere tutti i blocchi? No, continuo ad avere un accesso sequenziale a costo O(n)
    Soluzione: cache, copia in ram della tabella dei puntatori (l'accesso in ram è più veloce del disco!)

LEZIONE 48 (allocazione indicizzata, spazio libero) --> 16/12/2020

Allocazione non contigua

- Allocazione indicizzata
  Scopo:
  - mantenere tutti i puntatori relativi allo stesso file insieme invece che sparsi per la FAT
  - Evitare l'accesso sequenziale --> accesso diretto

  Caratteristiche:
  - Blocco dati
  - Blocco indice: array di puntatori che puntano ad altri blocchi (dati o indice)

  Esempio:
  Blocco indice = 512B
  Puntatori = 4B
  Res: in un blocco ci sono 512/4 = 128 puntatori

  Problema:
  - Nel momento in cui ho utilizzato *tutto* il blocco indice per creare dei puntatori ai blocchi dati, non posso più indicizzare nuovi blocchi dati!
  - Dimensione massima di un file = n.puntatori * dim.blocco (es. 128 * 512B = 64 KB) --> limitata!

  Soluzione:
  - Schema collegato: l'ultimo puntatore del blocco indice punta ad un altro blocco indice (simile a linked list)

  Vantaggio: file di dimensioni illimitate
  Problema: accesso sequenziale, se il file è estremamente grande, dovrò accedere in modo sequenziale a tutti i blocchi indice
  Soluzione: creo un blocco indice che punta ad altri blocchi indice (creo un albero a 2 o più livelli):
  - Livello 0: blocco indice
  - Livello 1: sotto-blocco indice
  - Livello 2: blocco dati

  Osservazioni:
  - Numero foglie = n.blocchi dati = (n.puntatori blocco indice)^livelli = 128^2 = 16.384 16K blocchi dati
  - Dimensione file massima = 16K * 512B = 2^23 MB = 8 MB

  Accesso diretto: quante letture di blocchi devo fare per avere il dato del blocco 'i'?
  - Lista collegata: 'i' blocchi
  - Albero: 3 letture (indipendentemente da 'i') --> costo O(logn)

  Struttura albero:
  - Dividiamo il file in blocchi dati (liv. 2)
  - Ogni blocco indice gestisce una serie di blocchi dati del file (liv. 1)
  - Un blocco indice gestisce i blocchi indice che gestiscono i blocchi dati (liv. 0)

  Come facciamo per identificare la posizione del byte da leggere? (simile a paginazione)
  Input: posizione di lettura del byte 'x', dimensione del blocco, dimensione del puntatore
  Output: indice del blocco indice da seguire
  Passaggi:
  - Qual è il blocco che contiene il byte 'x'? --> x / dim.blocco
  - Qual è l'offset all'interno del blocco? --> x % dim.blocco
  - Blocco indice [blocco] = riferimento al blocco dati
  - Blocco dati [offset] = byte

  Come facciamo per identificare l'indice del livello 0?
  Passaggi:
  - Livello 0: x / (dim.blocco)^2 = sottoalbero = indice
  - Livello 1: x - x / (dim.blocco)^2 = (x % (dim.blocco)^2) / dim.blocco

  Considerazioni:
  - File piccoli (contro)
  - Poche letture (1 per blocco indice + 1 per blocco dati) (pro)

  Idea: scelgo di usare e riempire strutture sempre più grandi in base alla dimensione del file; qesto mi permette di adattarmi alla situazione e quindi dare un'ottima capacità di memorizzazione con ottime prestazioni.

  Come:
  - Ho dei puntatori direttamente a dei blocchi, nel FCB
    - Se il file è estremamente piccolo (pochi byte), allora non utilizzo il blocco indice
    - Se il file è grande, allora prevedo uno o più contatori ad uno schema indicizzato a livelli

  Vantaggi:
  - File di dimensioni libere
  - Tempi di accesso proporzionali alle dimensioni del file; costo O(logn)

Spazio libero
Qual è il prossimo spazio libero che posso usare in allocazione?
Idee:

- Mappa bit (Bitmap): array con indice numero blocco (0=libero, 1=occupato)
- Lista collegata di blocchi liberi: man mano che voglio allocare qualcosa, prelevo e consumo dalla lista e aggiorno il puntatore --> costo O(1)
- Lista collegata tra diversi "Run" (sequenze continue di blocchi liberi)

Quanti blocchi può contenere un disco?
Esempio: disco da 1TB, dimensione blocco da 512B:
Blocchi: 1TB/512B = 2^40B/29B = 2^31 = 2G
Dimensione Bitmap: 2^31 * 1bit = 2^31/2^3B = 2?28B = 256MB