Cose che chiede al 100 per cento:

- memoria virtuale, segmentazione e paginazione, stallo, mutua esclusione, PCB, differenze tra threads e processi, schedulazione, stati dei processi, bisogna sapere bene i nomi degli algoritmi(dekker, banchiere…), chiede come funziona l'algoritmo del banchiere;

- chiede spesso i processi;

- domanda tipica del prof: cosa è la paginazione dinamica;

- chiede PCB, mutua esclusione, semafori e due cosette sui messaggi.

- chiede FIFO, LRU, optimal

- chiede politiche di scheduling e gli algoritmi di scheduling

- chiede politiche di schedulazione deil disco

questi argomenti erano ciò che aveva chiesto agli esami passati, poi al mio esame ha chiesto esattamente le stesse identiche cose, facendo le stesse identiche domande.

📌 SISTEMA OPERATIVO

🔹 1. Cos'è un Sistema Operativo?

Un **Sistema Operativo (OS)** è un **software di sistema** che agisce da intermediario tra l'hardware del computer e l'utente (o i programmi). Ha tre obiettivi principali:
✅ **Facilitare l'uso dell'hardware** → Rende più semplice usare il computer.
✅ **Essere efficiente** → Usa bene le risorse disponibili.
✅ **Evolversi** → Deve poter migliorare nel tempo (aggiungere nuove funzioni, correggere errori).

🔹 2. Struttura di un Computer (Livelli Principali)

Un computer è formato da diversi componenti organizzati in livelli:

🖥️ **LIVELLO 1: HARDWARE** (Componenti fisici)

- **Processore (CPU)** → Esegue i calcoli e controlla le operazioni.
- **Memoria principale (RAM)** → Memoria veloce che conserva dati temporaneamente.
- **Dispositivi di I/O** → Tastiera, schermo, mouse, stampante, ecc.
- **Bus (Interconnessioni di sistema)** → Collega CPU, memoria e dispositivi di I/O.

🖥️ **LIVELLO 2: SISTEMA OPERATIVO** (Software di base)

- Gestisce l'hardware e permette ai programmi di funzionare.
- Funziona come un "**traduttore**" tra hardware e software.

🖥️ **LIVELLO 3: LIBRERIE/UTILITY** (Strumenti avanzati)

- Offrono funzioni extra ai programmatori per creare software più facilmente.

🖥️ **LIVELLO 4: APPLICAZIONI** (Programmi usati dagli utenti)

- Es. browser, videogiochi, programmi di scrittura.

🔹 3. Perché il Sistema Operativo è Necessario?

Senza Sistema Operativo, il computer **non sarebbe utilizzabile** perché:

- L’hardware da solo non può comunicare con l’utente.
- I programmi devono avere un modo standard per interagire con il computer.

✅ Il sistema operativo (OS) **fornisce un'interfaccia tra hardware e software** e permette ai programmi di funzionare.

🔹 4. Funzioni Principali del Sistema Operativo

1️⃣ Creazione ed Esecuzione di Programmi

- 📌 Aiuta gli utenti a creare, scrivere e avviare programmi.
- 📌 Contiene un compilatore che trasforma il codice sorgente in un eseguibile.
- 📌 Carica i programmi nella memoria (RAM) per l'esecuzione.
- 📌 Gestisce i processi, ossia i programmi in esecuzione.

2️⃣ Gestione Input/Output (I/O)

- 📌 Controlla dispositivi come mouse, tastiera, monitor, stampante, ecc.
- 📌 Garantisce che i dati vengano trasferiti correttamente tra il computer e le periferiche.
- 📌 Fornisce un'interfaccia standard per l'uso dei dispositivi.

3️⃣ Gestione della Memoria

- 📌 Decide come distribuire la RAM tra i vari programmi per ottimizzare le prestazioni.
- 📌 Utilizza la memoria virtuale per aumentare lo spazio disponibile, caricando solo le parti necessarie del programma in memoria.

4️⃣ Gestione dei File (File System)

- 📌 Organizza e gestisce file e cartelle sul disco.
- 📌 Permette di creare, salvare, leggere, modificare ed eliminare file.
- 📌 Assegna automaticamente settori sul disco a ogni file, semplificando l'interazione dell'utente con il supporto di memorizzazione.

5️⃣ Comunicazione tra Processi

- 📌 Consente ai programmi in esecuzione di scambiarsi dati e informazioni, facilitando la collaborazione tra processi.

6️⃣ Gestione dei Processi

- 📌 Assegna risorse come CPU, memoria e dispositivi ai processi in esecuzione.
- 📌 Permette il multitasking, ovvero l'esecuzione contemporanea di più programmi.

7️⃣ Controllo degli Utenti

- 📌 Garantisce che solo utenti autorizzati possano accedere al sistema, proteggendo le risorse da accessi non autorizzati.

8️⃣ Gestione degli Errori

- 📌 Rileva problemi hardware o software, cercando di risolverli automaticamente o avvisando l'utente in caso di necessità.

9️⃣ Protezione e Sicurezza

- 📌 Impedisce accessi non autorizzati ai dati e alle risorse del computer.
- 📌 Utilizza metodi di autenticazione (come password, PIN, ecc.) per proteggere i dati degli utenti.

📢 IN CONCLUSIONE...

Il **Sistema Operativo** è il cuore del computer 💻:

- 🔹 Senza di esso, l'hardware non sarebbe utilizzabile.
- 🔹 Organizza e gestisce **memoria, dispositivi, file e programmi**.
- 🔹 Garantisce **sicurezza, efficienza e prestazioni ottimali**.

Chiarimento:
Nel testo di prima con "**disco**" si intende il **supporto di memorizzazione fisico** (quindi hard disk, SSD o altri dispositivi di memorizzazione persistente).

📌 GESTIONE DELLE RISORSE

- 🔹 Le **risorse** sono dispositivi come **monitor, stampanti, tastiera**.
- 🔹 Ogni dispositivo ha un **controller** che lo collega al sistema.
- 🔹 Alcuni dispositivi hanno un **loro Sistema Operativo**.
- 🔹 I **controllers** gestiscono lo **scambio di dati** tra dispositivo e sistema.
- 🔹 Il **processore** esegue le istruzioni, distinguendo tra:

✅ **Istruzioni del Sistema Operativo** (gestione del sistema).
✅ **Istruzioni dei programmi** (esecuzione delle attività).

- 🔹 Il **Sistema Operativo assegna il processore (CPU)** ai programmi in esecuzione.

📌 STORIA DEI SISTEMI OPERATIVI

1️⃣ Primi calcolatori (a valvole)

⚠️ **Controllo manuale**, senza sistema operativo.
⚠️ **Problemi di schedulazione**:
❌ Bisognava **aspettare** la fine di un programma prima di avviarne un altro.
❌ **Tempi molto lunghi**.

2️⃣ Arrivo dei transistor

💡 I transistor **controllano il passaggio di corrente** nei circuiti.
💡 **Programmi eseguiti in sequenza automaticamente** (senza intervento umano).
💡 Nasce il **"monitor"** (insieme di istruzioni per migliorare i programmi).
💡 Introduzione delle **interruzioni**:
✅ Permettono di **fermare** momentaneamente un programma per farne eseguire un altro più urgente.
✅ Prima, per interrompere un programma, bisognava **staccare la spina**! ⚡😲

3️⃣ Memoria condivisa e gestione delle interruzioni

📈 **Aumento dell'efficienza** dei sistemi.
📈 Introduzione della **multiprogrammazione**:
🔄 **Più programmi vengono eseguiti simultaneamente** per **ottimizzare** i tempi di attesa.
🔄 **Esempio**: Quando il processore è occupato solo per il **4% del tempo**, gli altri programmi vengono fatti avanzare.
🔄 Così si gestiscono meglio **memoria, processi e alternanza dei programmi**.

4️⃣ Personal computer e circuiti integrati

💻 Nascono i **computer personali (PC)** con circuiti integrati.
💻 Ora i computer sono **accessibili a tutti**.
💻 Inizio dell'era del **computing su larga scala**.

5️⃣ Multi-programmazione e time-sharing

⚡ **Multi-programmazione**:
✅ Obiettivo: Usare al massimo il processore, riducendo i tempi morti.
⚡ **Time-sharing**:
✅ Obiettivo: Dare l'illusione che il computer faccia più cose insieme, per migliorare l'esperienza dell'utente.

🖥️ IL SISTEMA DI ELABORAZIONE (è l'intero insieme di hardware)

🔹 I Componenti Principali

1. **CPU (Central Processing Unit)** → Esegue le istruzioni dei programmi.
2. **Memoria principale (RAM)** → Memorizza temporaneamente dati e istruzioni.
3. **Dispositivi di I/O (Input/Output)** → Tastiera, mouse, monitor, stampante, ecc.

🔹 **La CPU (PROCESSORE CENTRALE) è composta da:**

- 🔸 **ALU (Arithmetic Logic Unit)** → Fa calcoli e operazioni logiche.
- 🔸 **Registri** → Memoria velocissima dentro la CPU, usata per dati temporanei.

  I Registri interni alla CPU sono:
  - 🔸 **PC (Program Counter)** → Indica la prossima istruzione da eseguire.
  - 🔸 **IR (Instruction Register)** → Contiene l'istruzione attualmente in esecuzione.
  - 🔸 **MAR (Memory Address Register)** → Indica l'indirizzo di memoria da cui leggere o su cui scrivere i dati.
  - 🔸 **MBR (Memory Buffer Register)** → Contiene i dati letti o scritti in memoria.
  - 🔸 **I/O AR e I/O BR** → Sono usati per gestire l'indirizzamento e il trasferimento dei dati tra la CPU e i dispositivi di input/output.

🔹 **Memoria e Processori Specializzati**

🔹 **Cache** → Memoria velocissima accanto alla CPU, usata per conservare temporaneamente i dati e le istruzioni usati spesso.

🔹 **RAM (Random Access Memory)** → È una memoria molto veloce, cioè il computer può leggere e scrivere dati su di essa in pochissimo tempo (non salva i dati quando il PC si spegne).

🔹 **GPU (Graphics Processing Unit)** → Processore fatto apposta per gestire le immagini, capace di svolgere molte operazioni allo stesso tempo, ed è utile anche per altri compiti oltre alla grafica.

🔹 **DSP (Digital Signal Processing)** → Processore che gestisce audio, video e crittografia.

🔹 **Il Ciclo di Esecuzione della CPU (Fetch-Execute Cycle)**

💡 **È il processo con cui la CPU esegue le istruzioni, ripetendolo in continuazione.**

🔄 **Fasi del ciclo:**

1️⃣ **Fetch (Prelievo)** → La CPU prende l'istruzione dalla memoria RAM.
2️⃣ **Decode (Decodifica)** → Analizza l'istruzione per capire cosa deve fare.
3️⃣ **Execute (Esecuzione)** → L'ALU o un'altra unità esegue l'operazione richiesta.
4️⃣ **Repeat (Ripeti)** → Il Program Counter (PC) passa alla prossima istruzione e il ciclo ricomincia.

🔹 **Le Interruzioni (Interrupts)**

🔔 Le **interruzioni servono quando la CPU deve gestire eventi esterni** (es. input da tastiera, stampante pronta, operazione terminata, ecc.).

📌 **Come funziona un'interruzione?**

1️⃣ La CPU controlla se ci sono segnali di interruzione.
2️⃣ Se c'è un'interruzione, la CPU **sospende il programma attuale** e salva lo stato del lavoro.
3️⃣ Viene eseguita l'operazione richiesta (es. leggere un input, stampare un documento).
4️⃣ Dopo la gestione dell'interruzione, la CPU **riprende da dove si era fermata**.

💡 Le **interruzioni servono per non sprecare tempo di CPU** mentre aspetta un'operazione lenta (es. lettura da tastiera o da memoria esterna).

🔹 **Esempio di Esecuzione con Interruzione**

✅ 1. La CPU esegue un programma.
✅ 2. Un utente preme un tasto (arriva un'interruzione).
✅ 3. La CPU salva il suo stato e gestisce l'input.
✅ 4. Finita l'interruzione, la CPU riprende l'esecuzione del programma.

🔹 **Importante!**

- **Se un'interruzione arriva mentre un'altra è in corso**, la CPU aspetta che la prima sia completata prima di gestire la seconda.
- **Il sistema operativo** si occupa di gestire le interruzioni in modo efficiente.

📌 ARCHITETTURA DEI SISTEMI OPERATIVI

1️⃣ Cos'è un processo?

💡 Il concetto di **processo** può sembrare astratto, ma in realtà è abbastanza semplice: è fondamentalmente un **programma in esecuzione**. Vediamo nel dettaglio cosa significa:

1. **Programma vs. Processo:**
    - **Programma:** È un insieme di istruzioni scritte (ad esempio, in un file eseguibile) che da solo è solo un codice "statico".
    - **Processo:** Quando questo programma viene avviato ed eseguito, diventa un processo. In questo stato, il sistema operativo gli assegna risorse (come memoria, tempo CPU, file aperti, ecc.) e lo gestisce attivamente.
2. **Perché il concetto di processo è importante:**
    - **Esecuzione Contemporanea:** Il computer spesso esegue più attività (processi) contemporaneamente, anche se in realtà il processore esegue istruzioni una alla volta. Il sistema operativo, grazie al multitasking, alterna l'esecuzione dei vari processi in modo tale da dare l'impressione che siano eseguiti in parallelo.
    - **Isolamento e Gestione:** Ogni processo è isolato dagli altri. Questo isolamento permette di gestire le attività in modo indipendente: se un processo si blocca o ha problemi, questo non necessariamente compromette gli altri.
    - **Priorità e Sincronizzazione:** Il sistema operativo stabilisce quali processi devono avere maggiore accesso alle risorse (ad esempio, dare priorità a processi interattivi rispetto a quelli in background) e coordina il loro avanzamento per evitare conflitti e garantire che tutto funzioni in maniera ordinata.

    In Sintesi:

- **Processo = Programma + Stato in Esecuzione:** È un'entità attiva che contiene il codice del programma, i dati, lo stato corrente (come il punto in cui si trovava nell'esecuzione) e le risorse assegnate dal sistema.
- **Gestione delle Attività:** Il concetto di processo permette al sistema operativo di suddividere il lavoro, assegnare risorse in maniera efficiente e gestire il multitasking in modo che ogni attività "avanzi nel modo giusto".

2️⃣ Come funziona la memoria?

💡 Quando un programma diventa un processo, il sistema operativo **assegna una porzione della memoria RAM** a quel processo. Quindi per poter funzionare, un processo ha bisogno che il suo codice e i suoi dati siano **caricati in memoria (RAM)**. Vediamo alcuni aspetti fondamentali:

- **Caricamento in Memoria:**
  Quando un processo viene creato, il sistema operativo carica il programma dalla memoria di massa (come il disco rigido) nella RAM. Questo include:
  - Il **codice eseguibile** del programma.
  - Le **variabili e i dati** necessari durante l'esecuzione.
  - Le strutture per gestire la **stack** (per le chiamate di funzione) e l'**heap** (per la memoria dinamica).
- **Gestione della RAM:**
  La RAM è una risorsa limitata e preziosa. Per questo:
  - Il sistema operativo gestisce la memoria in modo efficiente, allocando solo la quantità necessaria per ciascun processo.
  - **Non tutto il programma viene sempre caricato contemporaneamente**: tecniche come il **paging** permettono di caricare in memoria solo le parti del programma che sono effettivamente necessarie in un dato momento.
  - La memoria è divisa in **aree specifiche** (come il codice, i dati, la stack, e l'heap) per ottimizzare l'esecuzione e l’accesso rapido ai dati.

Chiarimento:
Un **processo non può esistere** e funzionare **senza che il suo programma e i dati** vengano opportunamente **gestiti e allocati in memoria** dalla CPU e dal sistema operativo.

3️⃣ Protezione e sicurezza della memoria

💡 Dentro la memoria ci sono programmi e processi.
🔹 Il programma usa memoria per:
✅ **Il codice** (le istruzioni)
✅ **I dati**
✅ **Le variabili**
🔹 I programmi leggono e scrivono nella memoria, ma solo nel loro spazio.
🔹 Serve un sistema di **protezione della memoria** per evitare che:

❌ Un programma acceda a spazi di altri programmi.
❌ Qualcuno rubi o distrugga informazioni.

4️⃣ Gestione delle risorse

💡 Il computer ha molte risorse da condividere tra i processi.
🔹 **Il processore (CPU)** è la risorsa principale, perché esegue i programmi.
🔹 I programmi devono essere organizzati bene:
✅ Alcuni devono avanzare più velocemente.
✅ Altri possono aspettare di più.
🔹 Il sistema operativo gestisce anche:
✅ **RAM (Memoria volatile),** è molto veloce, ma i dati vi rimangono solo finché il computer è acceso.
✅ **Disco (Memoria di massa),** più lenta della RAM, ma conserva i dati anche quando il computer è spento.
✅ **Dispositivi (mouse, tastiera, ecc.),** questi sono gli strumenti con cui l'utente interagisce con il computer.

🔹 **Riassunto:** Il sistema operativo organizza processi, memoria, sicurezza e risorse per far funzionare tutto correttamente.

📌 COME IL SISTEMA OPERATIVO GESTISCE I PROGRAMMI?

💡 Chiamate di Sistema (System Calls)

- Sono **richieste speciali** fatte dai programmi al sistema operativo.
- 📌 Esempio: un programma che vuole aprire un file usa una **chiamata di sistema**.

🔹 Interfaccia Utente (Come l'Utente Interagisce con il PC?)

1️⃣ 🖥️ **Interfaccia Grafica (GUI)**
📌 Usa **icone, finestre e pulsanti** (Esempi: Windows, macOS).
2️⃣ ⌨️ **Linea di Comando (CLI)**
📌 L'utente scrive **comandi testuali** (Esempi: Linux, MS-DOS).

🔹 Gestione dei Processi

📌 Un **processo** è un programma in esecuzione.
📌 Il sistema operativo permette a più processi di funzionare insieme.

🟢 **Tipi di Processi:**
✅ **Indipendenti** → Non comunicano tra loro.
✅ **Interagenti** → Scambiano dati tra loro.

🔹 **Gestione della Memoria**

📌 La memoria RAM viene assegnata ai programmi in esecuzione.
📌 Il sistema operativo impedisce che un programma **usi troppa memoria**.

📌 **CONCLUSIONI**

Il sistema operativo è essenziale perché:
✅ Fa funzionare il computer.
✅ Permette di usare programmi e dispositivi.
✅ Gestisce sicurezza, risorse e memoria.

📌 SISTEMI OPERATIVI: INTERFACCE E CHIAMATE DI SISTEMA + STRUTTURA

1. CLI (Command Line Interface) - Interfaccia a riga di comando

✔️ Permette di eseguire comandi scritti dall'utente.
✔️ Può essere usata per **automatizzare attività** con script.
✔️ Offre **più controllo** rispetto alla GUI.
❌ **Non è intuitiva**, bisogna conoscere i comandi.

2. GUI (Graphical User Interface) - Interfaccia grafica

✔️ Interazione visiva con **icone, finestre e menu**.
✔️ **Facile da usare**, anche con touchscreen.
❌ Meno potente della CLI per alcune operazioni avanzate.

3. API (Application Programming Interface) - Interfaccia di programmazione

✔️ Permettono ai **programmi** di comunicare con il sistema operativo.
✔️ Una libreria di supporto aiuta a **tradurre** le API in chiamate di sistema.
✔️ Ogni chiamata di sistema ha **parametri** che possono essere passati in tre modi:

- **Registri CPU** → Dati inviati direttamente al processore.
- **Blocchi di memoria** → Dati salvati in memoria e letti dal sistema operativo.
- **Stack (pila di dati)** → Dati inseriti in una struttura LIFO (Last In, First Out).

4. Chiamate di sistema (categorie principali)

🖥️ **Controllo dei processi** → Creazione, esecuzione e arresto dei programmi.
📂 **Gestione dei file** → Creazione, apertura, modifica e cancellazione dei file.

📡 **Gestione dispositivi I/O** → Lettura e scrittura su dispositivi come tastiera e disco (memoria di massa).
⏳ **Gestione informazioni** → Ottenere ora, data e altre informazioni di sistema.
📶 **Comunicazione tra processi** → Connessioni, invio e ricezione di messaggi.
🔒 **Protezione e sicurezza** → Controllo accessi, autorizzazioni e gestione risorse.

5. Programmi di sistema

✔️ Sono programmi che aiutano a **gestire il sistema operativo e l'hardware**.
✔️ Servono per **controllare dispositivi**, file e configurazioni.
✔️ Possono essere:

- **Semplici** → Offrono solo le funzioni essenziali.
- **Avanzati** → Più veloci e completi, ideali per sistemi moderni.

6. Servizi del sistema operativo
Il sistema operativo offre una serie di servizi che aiutano le applicazioni e gli utenti a interagire con l'hardware. Questi servizi si possono suddividere in due categorie:

🔧 **Meccanismi** → **Funzioni di base** (es. accesso ai file e gestione della memoria).
📊 **Politiche** → **Strategie** per decidere quando e come eseguire i meccanismi (queste strategie sono le decisioni che il sistema prende).

7. STRUTTURA DEL SISTEMA OPERATIVO:

🏗️ **Organizzazione a livelli** → **approccio architetturale generale**: molti sistemi operativi adottano un'architettura a livelli o a moduli, in cui ogni livello o modulo è responsabile di determinate funzionalità. Questo permette di isolare le responsabilità e semplificare la manutenzione e l'aggiornamento del sistema.
Anche se c'è l'organizzazione a livelli si possono usare due stili diversi di progettazione del kernel all'interno di un sistema operativo:

- 💡 **Microkernel** → è uno **stile di architettura (più lento)** ADOTTATO SOLO DA **ALCUNI sistemi operativi**. È una struttura minima con solo le funzioni essenziali per far funzionare il sistema (come la comunicazione tra processi), il resto viene gestito da programmi esterni.
- 💡 **Kernel monolitico** → molti sistemi operativi adottano questo **stile di architettura (più veloce)**, gestisce molte funzionalità direttamente (come la gestione della memoria, dei filesystem e dei driver).

Chiarimento: Il **kernel** è il nucleo del sistema operativo, ovvero la parte fondamentale che si occupa di gestire e coordinare le risorse hardware e software del computer. In altre parole, è il "mediatore" tra le applicazioni e l'hardware.

📌 STRUTTURA DEL SISTEMA OPERATIVO + CREAZIONE

✅ 1. Sistemi Monolitici

- 🔹 Tutti i componenti (memoria, file, input/output, ecc.) sono in **un unico grande programma**.
- 🔹 **Vantaggi**: ✅ **Veloce** 🚀
- 🔹 **Svantaggi**: ❌ **Difficile da modificare e aggiornare** 😓
- 🔹 Se c'è un errore in un punto, può bloccare tutto il sistema ❗
- 🔹 📌 **Esempio**: Vecchi sistemi **UNIX**

✅ 2. Sistemi Stratificati

🟢 Il sistema è **diviso in livelli** 📊
🟢 Ogni livello ha **una funzione specifica** e comunica solo con i livelli vicini 🔄
🟢 **Vantaggi**: ✅ **Più facile da gestire e modificare** 🛠️
🔴 **Svantaggi**: ❌ **Può essere più lento** 🐢 perché le richieste passano attraverso più livelli

---

✅ 3. Microkernel

🟢 **Solo le funzioni essenziali** (memoria e comunicazione tra processi) sono nel **kernel** 🔑
🟢 Tutte le altre funzioni (gestione file, driver, ecc.) girano separatamente 🏗️
🟢 **Vantaggi**:
✅ **Più sicuro** 🔒 (se un modulo ha problemi, il resto funziona comunque)
✅ **Più facile da aggiornare** 🔄
🔴 **Svantaggio**: ❌ **Può essere più lento** 🐢 rispetto ai monolitici

---

📌 Debugging (Gestione degli Errori)

- 🔹 Capacità del sistema operativo di **individuare e correggere errori** 🧐
- 🔹 Molto importante quando più applicazioni lavorano insieme 🤝
- 🔹 **Difficoltà**: ❌ Richiede **strumenti avanzati** e può essere **complicato da gestire** 🧩

🔹 **Indipendentemente dallo stile architetturale scelto**, un buon supporto per il debugging facilita la manutenzione e l'aggiornamento del sistema.

📌 Come si crea un sistema operativo?

🟢 1. Scrivere il codice sorgente

✍️ Programmatori scrivono il codice in **C o Assembly** 📜
📌 Può essere **open-source** (codice accessibile a tutti) o **proprietario** 🔒

🟢 2. Compilare il codice

💾 Il codice viene trasformato in un **programma eseguibile** 🖥️

🟢 3. Configurare il sistema

🛠️ Si adatta il sistema al computer su cui verrà installato
📌 Include **driver**, impostazioni e parametri vari ⚙️

🟢 4. Installare il sistema operativo

💾 Viene installato su un **disco rigido o altro supporto di memoria** 🖴

🟢 5. Avviare il sistema operativo

🚀 Un piccolo programma chiamato **bootstrap loader** 📥 si trova nella **ROM (Read-Only Memory)**
⚡ Il bootstrap loader **carica il kernel** del sistema operativo e lo avvia 🖥️

Chiarimento:

la **ROM (Read Only Memory):**

- È una **memoria non volatile**, il che significa che conserva i dati anche quando il computer è spento.
- Viene utilizzata principalmente per memorizzare firmware o istruzioni di base (ad esempio, il BIOS o UEFI) che servono per l'avvio del sistema.
- I dati nella ROM sono generalmente scritti una sola volta (o molto raramente aggiornati) e non sono destinati a essere modificati frequentemente dall'utente o dal sistema operativo.

📌 PROCESSI NEI SISTEMI OPERATIVI (chiesti spesso dal prof)

1️⃣ Cos'è un processo?

Un **processo** è un **programma in esecuzione**.
🔹 Il processore può eseguire solo **un processo alla volta**, ma il sistema operativo gestisce più processi alternandoli.

✅ Differenza tra Programma e Processo

✔️ **Programma** → È solo un insieme di istruzioni, **non viene eseguito da solo**.
✔️ **Processo** → È il programma **in esecuzione**. Esiste solo per un certo periodo.

2️⃣ Struttura di un processo

Ogni processo ha diverse **parti fondamentali**:
📍 **Program Counter (PC)** → Tiene traccia dell'**istruzione successiva** da eseguire.
📍 **Registri della CPU** → Memorizzano i dati della CPU mentre il processo è attivo.
📍 **Stack** → Contiene dati temporanei (**variabili locali, indirizzi di ritorno**).

📍 **Heap** → Memoria usata per dati creati dinamicamente (**può espandersi**).
📍 **Dati e Codice** → Contengono le istruzioni e i dati globali del programma.

3️⃣ Stati di un processo

Un processo può **cambiare stato** durante la sua esecuzione:

🔵 **New (Nuovo)** → Il processo è in **fase di creazione**.
🟡 **Ready (Pronto)** → Il processo **aspetta la CPU** per essere eseguito.
🟢 **Running (In esecuzione)** → Il processo **sta usando la CPU**.
🟠 **Waiting (In attesa)** → Il processo è **sospeso**, in attesa di un evento esterno (es. input/output).
🔴 **Terminated (Terminato)** → Il processo ha **finito** e le sue risorse vengono liberate.

4️⃣ PCB (Process Control Block) – Descrittore di Processo

📌 Il **sistema operativo** usa una struttura speciale chiamata **PCB** per **tenere traccia dei processi**.

📂 Il PCB contiene:
✔️ **Lo stato** del processo (Pronto, In esecuzione, In attesa).
✔️ **Il PC (Program Counter)** e i **registri** della CPU per poter riprendere il processo dallo stesso punto.
✔️ **Le informazioni sulla memoria** e sulle operazioni di **input/output**.

📝 **Cosa succede quando un processo viene sospeso?**
➡️ Il suo **stato viene salvato nel PCB**.
➡️ Quando riprende, i dati vengono **ricaricati** e il processo continua da dove si era fermato.

5️⃣ Code di Schedulazione dei Processi

Il sistema operativo organizza i processi in **diverse code**, in base al loro stato:

📌 **Job Queue** → Contiene **tutti** i processi del sistema.
📌 **Ready Queue** → Contiene i processi **pronti per l'esecuzione**.
📌 **Device Queue** → Contiene i processi in **attesa di input/output**.

6️⃣ Tipi di Schedulatore

Il sistema operativo decide l'**ordine in cui i processi** vengono **eseguiti grazie a tre schedulatori**:

✅ Schedulatore a breve termine (CPU Scheduler)

- Sceglie il processo pronto che verrà eseguito subito.
- Questa operazione è **molto frequente**.

✅ Schedulatore a lungo termine

- Decide quali processi entrano nella coda dei processi pronti.
- Questa operazione è **meno frequente**.

✅ **Schedulatore a medio termine**

- Può sospendere un processo per liberare memoria.
- Usato **nei sistemi con caricamento dinamico della memoria**.

🎯 **Ricapitolando:**

📌 Un **processo** è un programma in esecuzione.
📌 Esistono **5 stati** di un processo (New, Ready, Running, Waiting, Terminated).
📌 Il **PCB** salva tutte le informazioni di un processo.
📌 Il sistema operativo gestisce i processi con **code e schedulatori**.

📌 GESTIONE DEI PROCESSI (chiesto spesso dal prof)

✅ 1. Selezione dei Processi

Il sistema operativo sceglie quale processo può usare la CPU. Esistono due tipi di processi:

🔹 **Processi I/O bound**
➡️ Si interrompono spesso perché aspettano operazioni esterne (es. lettura da disco).

🔹 **Processi CPU bound**
➡️ Si interrompono raramente e usano la CPU per molto tempo.

🎯 Obiettivi della selezione

✔️ Evitare che un solo processo usi tutta la CPU.
✔️ Garantire un buon equilibrio tra processi diversi.
✔️ Gestire tutto con una strategia a lungo termine.

⚠️ Overhead (tempo perso nei cambi di processo)

Cambiare da un processo all'altro fa perdere tempo. Questo dipende da:
🟢 **Velocità della memoria** → più veloce è il trasferimento dati, meno tempo si perde.
🟢 **Numero di registri del processore** → più registri ci sono, più grande è l'immagine del processo.
🟢 **Istruzioni speciali** → alcuni processori hanno comandi per velocizzare il cambio di processo.

📌 **L’obiettivo è bilanciare efficienza e controllo del sistema!**

✅ 2. Creazione e Terminazione dei Processi

🛠️ Creazione

✔️ Un processo può creare altri processi, detti **figli**.
✔️ I processi figli possono **condividere** le risorse del padre o avere **risorse proprie**.
✔️ Tutti i processi insieme formano una **gerarchia ad albero**.

🛑 Terminazione

✔️ Quando un processo termina, le sue risorse vengono **liberate**.
✔️ Se un processo padre termina, anche i figli possono terminare (**terminazione a cascata**).
✔️ Se un figlio termina prima del padre, diventa **un processo orfano** e il sistema operativo deve gestirlo.

🆔 Identificazione dei Processi

📌 Ogni processo ha un numero unico chiamato **PID (Process Identifier)**.

✅ 3. Esempio di Gerarchia di Processi (Linux)

📌 Struttura dell'albero dei processi

```
📂 init (PID = 1)
 ├── 🖥️ bash (PID = 916)
 │    ├── 📄 ls (PID = 1015)
 │    ├── 📄 cat (PID = 1018)
 │    ├── 🔑 passwd (PID = 200)
 ├── ⚙️ kthread (PID = 2)
 │    ├── ⚡ ksoftirqd (PID = 3)
```

📌 Cosa significa?

- **Gerarchia ad albero:**
  Ogni riga rappresenta un **processo identificato da un PID (Process ID).** I processi sono organizzati in una struttura ad albero dove ogni processo (eccetto il primo) ha un processo "genitore" che lo ha creato.
- **Processo "init":**
  È il **primo processo avviato dal sistema operativo** (con PID = 1) ed è il processo radice da cui derivano tutti gli altri processi.
- **Processi figli:**
  Ad esempio, il processo **bash** (PID = 916) è figlio di **init** e, a sua volta, crea altri processi come **ls**, **cat** e **passwd**. Questo rappresenta la relazione padre-figlio tra processi.
- **Ruolo dei processi:**
  - **bash:** È una shell che consente all'utente di interagire con il sistema.
  - **ls, cat, passwd:** Sono comandi eseguiti dalla shell.
  - **kthread e ksoftirqd:** Sono processi legati al kernel, responsabili di compiti specifici a livello di sistema.

🔎 **RICORDA:**
✅ La CPU deve essere assegnata in modo equilibrato.
✅ Troppi cambi di processo causano perdita di tempo (Overhead).
✅ I processi possono creare figli e terminare con una logica a cascata.
✅ Ogni processo ha un identificatore unico (PID).

📌 THREADS (chiesti spesso dal prof, in particolare la differenza tra thread e processo)

1️⃣ Differenza tra Thread e Processo

✅ **Processo** = Programma in esecuzione con una sua memoria separata.
✅ **Thread** = Parte di un processo che esegue operazioni indipendenti e condivide memoria e risorse.
✅ **Processi** → Isolati tra loro.
✅ **Thread** → Possono lavorare insieme dentro lo stesso processo.
🎯 **Vantaggio**: I thread permettono di eseguire più operazioni contemporaneamente dentro un processo.

2️⃣ Caratteristiche dei Thread

📌 **Condivisione delle risorse** → Tutti i thread di un processo condividono:
- 🔹 Memoria
- 🔹 File aperti
- 🔹 Variabili globali

📌 **Esecuzione indipendente** → Ogni thread può svolgere un compito diverso nello stesso momento.

⚠️ **Rischi di errore**:
❌ Se non programmati bene, i thread possono causare:
- 🔺 **Deadlock** (blocco del sistema) 🚫
- 🔺 **Sovraccarico della memoria** 🛑

📌 **Stati di un thread**:
🔄 **Pronto** → Aspetta di essere eseguito.
🏃 **In esecuzione** → Sta svolgendo il suo compito.
⏳ **In attesa** → Aspetta una risorsa (es. input).
✅ **Terminato** → Ha finito il suo compito.

🎯 **Vantaggio**: Creare e gestire un thread è più veloce rispetto a un processo.

3️⃣ Vantaggi del Multithreading

✅ **Uso efficiente del processore** → Se un thread è in attesa, un altro può lavorare.
✅ **Esecuzione parallela** → I thread possono lavorare insieme su più core della CPU.
✅ **Reattività** → Le applicazioni rispondono più velocemente agli input dell'utente.

🎯 **Esempio pratico**:
💬 Un'app di chat può usare un thread per ricevere messaggi e un altro per scrivere, evitando blocchi.

4️⃣ **Problemi del Multithreading**

⚠️ **Sincronizzazione** → Se più thread accedono agli stessi dati contemporaneamente, possono verificarsi errori.
📌 **Soluzione**: Usare **semafori** 🛑 per gestire l'accesso ai dati.

⚠️ **Debugging difficile** → Gli errori nei thread, come il **deadlock**, sono complessi da trovare e correggere.
⚠️ **Consumo di risorse** → Troppi thread possono rallentare il sistema invece di velocizzarlo.

5️⃣ **Tipi di Thread nel Sistema Operativo**

📌 **Thread a livello kernel** 🖥️
✅ Gestiti dal sistema operativo.
✅ Più **potenti**, ma **consumano più risorse**.

📌 **Thread a livello libreria (la gestione e le operazioni sui thread sono fornite tramite l'API della libreria)** 📚
✅ Gestiti da librerie senza coinvolgere direttamente il sistema operativo.
✅ Più **veloci**, ma con **meno controllo**.

Esistono **API** come **POSIX Threads (pthread)** che offrono funzioni per creare, gestire e sincronizzare i thread.

📌 **Thread a livello utente** 👤
✅ Gestiti direttamente dall'applicazione (i thread a livello utente vengono gestiti interamente dall'applicazione o da una libreria, e non dal kernel del sistema operativo. Questo significa che il sistema operativo non è a conoscenza dei singoli thread creati all'interno del processo).
✅ Più **efficienti**, ma **NON sfruttano bene i processori multipli** (vuol dire che, nonostante possano esistere più thread all'interno dell'applicazione, questi non possono essere eseguiti simultaneamente su processori o core diversi, limitando l'efficacia nel parallelizzare le operazioni).

🎯 **Curiosità**:
⚡ Alcuni sistemi operativi non supportano il **multiprocessing** con thread a livello utente.

📖 **Conclusione**

💡 Usare i thread è utile per migliorare le prestazioni, ma serve attenzione per evitare problemi come il deadlock!

📌 LEGGE DI AMDAHL E MULTITHREADING

🔢 **Formula della Legge di Amdahl**

Questa formula calcola il **massimo miglioramento di velocità quando un programma viene diviso su più processori**:

Speedup = 1 / ((1−f) + f / n)

🔹 **f** = parte del programma che può essere eseguita in parallelo (**tra 0 e 1**). Se fosse 1, tutto il programma sarebbe parallelo (ma non succede mai).
🔹 **n** = numero di processori usati.

⚡ Cosa succede aumentando i processori?

✅ **Più processori → Tempo di esecuzione minore** (ma mai a **zero**!).
❌ Se **f è bassa** (es. 0,5 o meno), aggiungere processori **non porta grandi vantaggi**.
❌ Aumentare troppo i processori aumenta **costi e difficoltà di gestione**.

💡 **Conclusione:** Il miglioramento ha un **limite** perché c'è sempre una parte del programma che non può essere parallelizzata.

🖥️ Quando il multithreading è davvero utile?

📌 **Applicazioni scritte per usare più thread**.
📌 **Pochi processi, ma con più thread** (es. videogiochi, programmi grafici, ecc.).
📌 **Programmi multiprocesso** (es. Java, perché la JVM usa più thread anche per una singola applicazione).

🛠️ Come funzionano i processi nei sistemi Unix?

📌 Il sistema crea un nuovo **task** con il comando **fork()**.
📌 **fork()** copia il processo attuale, incluse le sue variabili e la memoria.
📌 Il processo figlio può essere:

- **Una copia indipendente** con tutti i suoi dati.
- **Un clone** che condivide i dati con il processo padre.

🔍 Riepilogo rapido

✅ La Legge di Amdahl dice che **più processori riducono il tempo**, ma solo fino a un certo punto.
✅ Se il programma **non è abbastanza parallelo**, tanti processori **non servono**.
✅ Il **multithreading è utile** solo se il programma è fatto per sfruttarlo.
✅ Nei sistemi Unix, i nuovi processi si creano con **fork()**, che può fare copie o cloni.

📌 CONCORRENZA TRA PROCESSI (chiesta spesso dal prof)

🔹 1. Cos’è la **Concorrenza tra Processi**?

Quando più processi vengono eseguiti contemporaneamente e usano le stesse risorse, possono sorgere conflitti.

- **Esempi di concorrenza:**
  - **Applicazioni multiple:** Processi indipendenti che non si conoscono e competono per le risorse.
  - **Applicazioni strutturate:** Processi che interagiscono e condividono risorse, ma devono **coordinarsi** per evitare conflitti.

🔹 2. **Problemi della Concorrenza**

Quando più processi accedono a risorse condivise senza controllo, possono verificarsi problemi:

🔻 a) Race Condition (Condizione di Gara)

- Accade quando due processi leggono e scrivono su una risorsa condivisa in modo non sincronizzato (coordinato).
- Il risultato finale dipende dall’ordine di esecuzione e può essere sbagliato.

📌 Esempio:

- Due processi P1 e P2 leggono e scrivono sulla stessa variabile **A**.
- P1 legge **A** e la usa.
- P2 modifica **A** prima che P1 abbia terminato.
- P1 ottiene un valore errato e mostra un carattere sbagliato.

🔻 b) Deadlock (Blocco Totale)

- Due o più processi sono bloccati perché ciascuno sta aspettando una risorsa che è già occupata da un altro processo. A causa di ciò nessun processo può proseguire.

🔻 c) Starvation (Processo Affamato)

- Un **processo non riesce mai ad accedere alla risorsa necessaria** perché altri processi la monopolizzano.

🔹 3. **Soluzioni ai Problemi di Concorrenza:**

Per evitare problemi, bisogna controllare come i processi accedono alle risorse condivise.

✅ a) Sezione Critica

- Parte del codice in cui un processo usa una risorsa condivisa.
- Solo un processo alla volta deve poter entrare in questa sezione critica per evitare conflitti.

✅ a) Metodi per Gestire la Sezione Critica

1. 🔸 **Lock (Blocco):**
   - Un processo "blocca" la risorsa prima di usarla e la "libera" dopo aver terminato.
2. 🔸 **Istruzioni Atomiche:**

- Operazioni speciali del processore (es. **compare_and_swap**) che garantiscono un accesso **senza interruzioni** alla risorsa.

3. 🔸 **Algoritmi Software:**
   - Regole per garantire l'accesso ordinato alla risorsa senza bisogno di hardware speciale.

✅ b) Mutua Esclusione, Cosa è?

Quando un **processo usa una risorsa** (CPU, memoria, file, dispositivi I/O), **nessun altro** processo deve **poterla usare nello stesso momento**.

- Serve per evitare problemi quando più processi accedono a risorse condivise.

✅ b) Mutua Esclusione Senza Istruzioni Speciali

1. **Obiettivo**: Creare un algoritmo che garantisca la mutua esclusione senza l'uso di istruzioni hardware speciali (come **test-and-set**).
2. **Soluzione**:
   Utilizzare **flag** (indicatori) per segnalare se una risorsa è occupata. I processi si alternano nell'accesso alle risorse in base a questi flag.

✅ c) Algoritmo di Peterson (una soluzione per i problemi della mutua esclusione derivati dalla concorrenza)

Un metodo software che:
- **Evita il deadlock.**
- **Evita la starvation.**
- **Permette l'accesso equo alla risorsa.**

📌 Come funziona?

- Ogni processo usa un **flag** per dire che vuole entrare nella sezione critica.
- Una variabile `turn` decide quale processo può entrare per primo.
- Se un processo vede che l'altro è già dentro, aspetta finché non è il suo turno.

✅ d) Algoritmo di Dekker (una soluzione per i problemi della mutua esclusione senza istruzioni speciali derivati dalla concorrenza)

Utilizza **due variabili (flag)** per indicare l'intenzione di entrare nella sezione critica e **una variabile di turno** per alternare l'accesso.
Come funziona:
- **Flag**: Ogni processo imposta la propria flag a **vero** quando vuole entrare nella sezione critica.
- **Variabile di turno**: Ogni processo controlla la variabile di turno per determinare se deve **cedere il turno** all'altro processo.
- **Regola**: Un processo entra nella sezione critica solo quando:
  - La sua flag è **vera**.
  - L'altro processo ha la flag **falsa** o il suo turno è passato.

Obiettivo: Garantire che solo **un processo alla volta** entri nella sezione critica e che non ci siano **interferenze** tra i processi.

📝 **Riassunto Finale:**

- La **concorrenza** (più processi competono per le risorse del sistema) può causare problemi come **race condition**, **deadlock** e **starvation**.
    - Soluzioni ai problemi della concorrenza:
    - **Sezione Critica**: Solo un processo alla volta accede alle risorse.
    - **Lock**: Meccanismo per evitare accessi simultanei.
    - **Istruzioni Atomiche**: Operazioni senza interruzione.
    - **Mutua Esclusione:** impedisce che più processi usino una risorsa contemporaneamente.
    - **Mutua Esclusione Senza Istruzioni Speciali**: Utilizzo di flag per segnalare risorse occupate.
    - **Algoritmo di Peterson**: Usa flags e una variabile di turno per garantire l'accesso equo.
    - **Algoritmo di Dekker**: Usa flag e una variabile di turno per gestire l'accesso senza istruzioni speciali.

🔵 SEMAFORI (chiesto spesso dal prof) E MONITOR

✅ 1. Semafori

I **semafori** sono strumenti per gestire l'accesso di più processi a risorse condivise.

📌 **Tipi di semafori:**

🔹 **Semaforo Generico**

📌 **Permette di gestire più processi contemporaneamente.**
Ha due operazioni principali:

- 🔽 **semWait:**
    - ➖ **Diminuisce** il valore del semaforo.
    - ⏳ Se il valore diventa negativo, il processo si blocca in attesa.
- 🔼 **semSignal:**
    - ➕ **Aumenta** il valore del semaforo.
    - 🔄 Se ci sono processi in attesa, ne risveglia uno.

🔹 **Semaforo Binario**

📌 **Può assumere solo due valori: `0` (bloccato) e `1` (libero).**
🔒 Funziona come un **mutex** (blocco esclusivo di risorse).

- 🔽 **semWaitB:**
    - ✅ Se il semaforo vale **1**, lo porta a **0** e il processo procede.
    - ⏳ Se il semaforo è già **0**, il processo si blocca in attesa.
- 🔼 **semSignalB:**

- 🔄 Se ci sono processi in attesa, ne risveglia uno.
- 🔙 Se non ci sono processi in attesa, imposta il semaforo a **1**.

📌 **Perché i semafori sono utili?**

✔️ Sono **operazioni atomiche** (non possono essere interrotte).
✔️ Evitano **stallo** (deadlock) e **starvation** (processi bloccati per sempre).
✔️ Permettono ai processi di accedere ordinatamente alle risorse.

✅ **2. Monitor**

📌 **I monitor sono strutture avanzate per la gestione dell'accesso esclusivo alle risorse.**
🔴 Solo un **processo alla volta** può eseguire il codice dentro un monitor.

📌 **Come funziona un monitor?**

- 🔹 Le **variabili e le procedure** sono accessibili solo ai processi che hanno il controllo del monitor.
- 🔹 I **dati condivisi** non sono accessibili dall'esterno.
- 🔹 Un solo processo alla volta può eseguire il codice del monitor.

🔹 **Variabili di condizione**

📌 Se un **processo deve aspettare una certa condizione, il monitor lo sospende** fino a quando la condizione diventa vera.

Due operazioni fondamentali:

- ⏳ **wait(c):**
  - Il **processo si sospende in attesa** della condizione `c`.
- 🔄 **signal(c):**
  - Se c'è un **processo in attesa** sulla condizione `c`, **lo riattiva**.
  - ❌ Se nessuno è in attesa, non fa nulla.

📌 **Perché i monitor sono utili?**

✔️ **Gestiscono situazioni complesse**, come il problema **produttori-consumatori (lo vedi più avanti)**.
✔️ **Rendono sicura e ordinata la gestione delle risorse condivise.**
✔️ **Eseguono un solo processo alla volta**, evitando problemi di concorrenza.

🔥 RIEPILOGO VELOCE

⚙️ Strumento	🎯 A cosa serve?	🔄 Funzionamento	📌 Esempio d'uso
🔵 **Semaforo Generico**	Gestire più processi	`semWait` ↓ `semSignal` ↑	Controllo accesso a file
🔵 **Semaforo Binario**	Bloccare/sbloccare risorse (mutex)	`semWaitB` (0 o 1)	Protezione di sezioni critiche
🔴 **Monitor**	Gestire l'accesso esclusivo a risorse condivise	`wait(c)`, `signal(c)`	Sincronizzazione di produttori e consumatori

---

💡 COMUNICAZIONE TRA PROCESSI: SCAMBIO DI MESSAGGI (chiesto spesso dal prof)

✅ Cos'è lo scambio di messaggi?

Invece di condividere direttamente dati e memoria, i processi possono **scambiarsi messaggi** per comunicare.
📌 Questo metodo è utile nei **sistemi distribuiti**, dove condividere memoria è difficile.

📨 COME FUNZIONA LO SCAMBIO DI MESSAGGI?

Due operazioni principali:

1️⃣ **send(destinatario, messaggio)** → Invia un messaggio a un altro processo.
2️⃣ **receive(mittente, messaggio)** → Riceve un messaggio da un altro processo.

🔹 Queste operazioni possono essere:
✔️ **Bloccanti** → Il processo aspetta finché il messaggio non viene ricevuto.
✔️ **Non bloccanti** → Il processo continua a lavorare anche se il messaggio non è arrivato.

🔹 **Tipi di indirizzamento:**

- **Diretto** → Il messaggio viene inviato a un destinatario specifico.
- **Indiretto** → Il messaggio viene messo in una **mailbox** e il destinatario lo prende quando vuole.

📌 **La coda dei messaggi segue la regola FIFO** (First In, First Out → il primo messaggio inserito è il primo a essere letto).

🔄 PROBLEMA CLASSICO: PRODUTTORE-CONSUMATORE

Un **Produttore inserisce** dati in un buffer.
Un **Consumatore preleva** i dati dal buffer.

🚨 **Problemi possibili:**

❌ Il **buffer è pieno** → Il produttore deve aspettare.
❌ Il **buffer è vuoto** → Il consumatore deve aspettare.

🛠️ **SOLUZIONE CON SEMAFORI**

Usiamo due semafori per sincronizzare i processi:

- 🔹 **s** → Regola l'accesso al buffer.
- 🔹 **delay** → Blocca il consumatore quando il buffer è vuoto.

👨‍💻 **Algoritmo Produttore**

1️⃣ Produce un dato.
2️⃣ Se il buffer non è pieno, lo **aggiunge**.
3️⃣ Se il consumatore stava aspettando, lo **sblocca**.

👨‍💻 **Algoritmo Consumatore**

1️⃣ Controlla se il buffer è vuoto.
2️⃣ Se il buffer non è vuoto, **preleva** un dato.
3️⃣ Se il produttore stava aspettando, lo **sblocca**.

- 🔹 Il semaforo `delay` **evita che il consumatore cerchi dati** QUANDO il buffer è vuoto.

🔧 **SOLUZIONE CON MONITOR**

I **monitor** sono un meccanismo più avanzato che usa **variabili di condizione** per evitare errori di sincronizzazione.

📌 **Funzioni principali:**

- **append(dato)** → Aggiunge un dato al buffer, se c'è spazio.
- **take()** → Preleva un dato, se il buffer non è vuoto.

📌 **Vantaggi del monitor:**

✔️ Evita che più processi accedano contemporaneamente al buffer.
✔️ Garantisce un funzionamento più ordinato e sicuro.

📌 **CONCLUSIONE**

📨 Lo **scambio di messaggi** è utile nei sistemi distribuiti.
📦 Per il **problema Produttore-Consumatore**, possiamo usare:
✔️ **Semafori** → Per sincronizzare manualmente i processi.
✔️ **Monitor** → Per gestire automaticamente l'accesso al buffer.

🔥 Scegliere il metodo giusto aiuta a evitare errori e migliorare l’efficienza!

PROBLEMA LETTORI/SCRITTORI

1️⃣ Descrizione del problema lettori/scrittori

Abbiamo una **risorsa condivisa** (come un file o una variabile) a cui possono accedere due tipi di processi:

- 🟢 **Lettori** → possono leggere contemporaneamente senza problemi.
- 🔴 **Scrittori** → devono scrivere in esclusiva, senza altri lettori o scrittori.

⚠️ **Problema**: dobbiamo gestire correttamente l’accesso per evitare **conflitti** e **blocchi** tra i processi.

2️⃣ Strategie di gestione

Esistono due strategie principali per controllare l’accesso alla risorsa:

📌 Precedenza ai lettori

✅ I lettori possono leggere insieme.
⛔ Gli scrittori devono aspettare che tutti i lettori finiscano.
⚠️ Problema: se ci sono molti lettori, gli **scrittori** potrebbero **dover aspettare per sempre**.

📌 Precedenza agli scrittori

✅ Uno scrittore che arriva blocca i lettori finché non ha finito.
⛔ Se ci sono scrittori in attesa, i lettori non possono iniziare a leggere.
⚠️ Problema: i **lettori** potrebbero **dover aspettare per sempre** se ci sono sempre scrittori.

3️⃣ Soluzione con semafori (Precedenza ai Lettori)

Usiamo **semafori** per controllare l’accesso alla risorsa:

- **Variabili usate**:
  - `readcount`: **conta** quanti lettori stanno leggendo.
  - `wsem`: **blocca** gli scrittori quando i lettori sono attivi.
  - `mutex`: garantisce che `readcount` **sia aggiornato in modo sicuro**.
- **Funzionamento**:
  🟢 **Quando un lettore arriva**:
  - **Incrementa** `readcount`.
  - Se è il **primo** lettore, **blocca gli scrittori** (`wsem`).

  🔴 **Quando un lettore termina**:

- **Decrementa** `readcount`.
- Se è **l'ultimo** lettore, **sblocca gli scrittori** (`wsem`).

✍️ **Scrittore cosa fa**:

- **Aspetta** che `readcount` sia 0.
- Quando **finisce, sblocca** **`wsem`** per gli altri scrittori.

4️⃣ Soluzione con semafori (Precedenza agli Scrittori)

Per dare priorità agli scrittori, usiamo **più semafori**:

- **Variabili usate**:
  - **`readcount`**: numero di lettori attivi.
  - **`writecount`**: numero di scrittori in attesa.
  - **`x`, `y`, `z`**: semafori per coordinare l'accesso alla risorsa.
- **Funzionamento**:

  🟢 **Quando un lettore arriva**:
  - Se **non ci sono scrittori in attesa** (`writecount = 0`), **può leggere**.
  - Se **ci sono** scrittori in attesa, deve **aspettare**.

  🔴 **Quando uno scrittore arriva**:
  - **Blocca** l'accesso ai **lettori**.
  - **Dopo** aver scritto, **sblocca** i lettori.

  🚦 **Meccanismo di blocco**:
  - Se uno **scrittore** è **attivo**, **nessun lettore può entrare**.
  - Quando **non ci sono più scrittori**, i **lettori** riprendono il **controllo**.

5️⃣ Soluzione con Scambio di Messaggi

Un'**alternativa all'uso dei semafori** è un **processo controllore** che gestisce gli accessi alla risorsa:

- 📩 I **lettori** e gli **scrittori** inviano **una richiesta** al controllore.
- ⏳ **Aspettano** finché il controllore non dà il permesso di procedere.
- ✅ **Dopo** aver finito, inviano un **messaggio di completamento** al controllore.
- 🔁 Il **controllore decide** chi può accedere dopo.

Come funziona?

📤 **Lettore/Scrittore invia richiesta** → Controllore verifica chi può accedere → 📩 Controllore concede accesso → 📤 Lettore/Scrittore invia conferma di fine → 🔄 Il ciclo continua.

Vantaggi e Svantaggi

✅ Evita starvation, perché il controllore gestisce le priorità in modo bilanciato.
✅ Maggiore controllo sugli accessi.
❌ Più complesso da implementare rispetto ai semafori.

6️⃣ Conclusione

Il **problema lettori/scrittori** è un classico della programmazione concorrente.
Esistono **tre principali soluzioni**:

Strategia	Vantaggi	Svantaggi
Precedenza ai lettori	**Lettori** possono **leggere liberamente**	**Scrittori** possono **aspettare troppo**
Precedenza agli scrittori	**Scrittori** possono **scrivere senza interruzioni**	**Lettori** possono **aspettare troppo**
Scambio di messaggi	**Più controllo** sugli accessi	**Complesso** da implementare

📌 Quale scegliere?

- Se i lettori devono avere **risposte veloci** → **Precedenza ai lettori**
- Se gli scrittori devono avere **accesso rapido** → **Precedenza agli scrittori**
- Se serve **gestione bilanciata** → **Scambio di messaggi**

👉 **Conclusione**: la scelta dipende dall'applicazione e dal tipo di accesso alla risorsa!

📌 STALLO (DEADLOCK) (chiesto spesso dal prof)

Lo **stallo** è una situazione in cui **uno o più processi si bloccano** perché:

- ✅ Un processo è in attesa di una risorsa posseduta da un altro processo.
- 🚫 Nessun processo può avanzare.

🔍 Esempio di Stallo

1. **P1** ha bisogno delle risorse **A** e **B**.
2. ✅ P1 prende la risorsa **A**.
3. **P2** prende la risorsa **B**.
4. Ora:
    - **P1 aspetta B**.
    - **P2 aspetta A**.
5. ❌ Entrambi i processi restano bloccati.

📂 Tipi di Risorse

1. ♻️ **Riutilizzabili**:
   - Possono essere usate più volte (es. CPU, memoria).
2. 🗑️ **Consumabili**:
   - Spariscono dopo l'uso (es. messaggi, dati, segnali).

⚠️ Condizioni per il Deadlock

Il deadlock si verifica solo se sono vere **tutte e 4 queste condizioni**:

1. 🔒 **Mutua esclusione**:
   - Una **risorsa** può essere usata da **un solo processo** alla volta.
2. ⏳ **Blocco e attesa**:
   - Un **processo** che possiede **risorse sta aspettando altre** risorse.
3. 🚫 **Assenza di prelazione**:
   - Le **risorse non possono essere tolte forzatamente** da un processo.
4. 🔄 **Attesa circolare**:
   - C'è una catena di processi in cui **ogni processo aspetta una risorsa posseduta dal successivo**.

✅ Prevenzione dello Stallo

Per evitare il deadlock, basta **eliminare almeno una delle 4 condizioni**.

🛠️ Metodi di Prevenzione

1. **Indiretti**:
   - 🔒 **Mutua esclusione**: Eliminabile solo in alcuni casi.
   - ⏳ **Blocco e attesa**:
     - Imporre che un processo ottenga tutte le risorse necessarie in una sola volta.
   - 🚫 **Assenza di prelazione**:
     - Togliere risorse solo se sono salvabili e riutilizzabili.
2. **Diretti**:
   - 🔄 **Evitare l'attesa circolare**:
     - Assegnare le risorse in un ordine prestabilito.

🔧 Risoluzione dello Stallo

Se lo stallo si verifica, esistono due strategie principali:

1️⃣ Rifiuto di Esecuzione

🚫 Non si assegnano risorse se non è garantito che i processi possano terminare.

2️⃣ Algoritmo del Banchiere

L'algoritmo **verifica** se una **richiesta** di risorsa **può essere soddisfatta** senza causare uno stallo.

📊 Dati Richiesti

- **R**: Numero totale di ogni tipo di risorsa.
- **V**: Risorse attualmente disponibili.
- **C**: Massimo numero di risorse richieste da ogni processo.
- **A**: Risorse già assegnate a ogni processo.

🔎 Procedura

1. **Calcolo della Matrice Bisogno**:
   📌 $B = C - A$.
2. **Verifica dello Stato Sicuro**:
   - ✅ Trovare un processo che può essere completato con le risorse attuali.
   - ✅ Liberare le risorse del processo completato.
   - 🔄 Ripetere finché:
     - Tutti i processi sono completati.
     - Oppure scopriamo uno stallo.
3. **Conclusione**:
   - ✅ Se tutti i processi possono terminare, lo stato è **sicuro**.
   - 🚫 Altrimenti, rifiutare la richiesta.

📋 Esempio dell'Algoritmo del Banchiere

1. 🛠️ **Stato Iniziale**:
   - **R**: Risorse totali.
   - **V**: Risorse disponibili.
   - **C**: Risorse richieste.
   - **A**: Risorse già assegnate.
2. 🔢 **Eseguire i Calcoli**:
   - Determinare il **Bisogno**:
     📌 $B = C - A$.
   - Verificare se esiste un ordine sicuro per completare i processi.
   - Se un processo può terminare, liberare le sue risorse.
3. 🛑 **Conclusione**:
   - ✅ Se lo stato è sicuro, la richiesta è accettata.
   - 🚫 Altrimenti, la richiesta è rifiutata.

📝 Riassunto Finale

- ✅ Lo stallo si verifica solo se sono vere tutte e 4 le condizioni (mutua esclusione, blocco e attesa, assenza di prelazione, attesa circolare).
- 🔄 Si può prevenire eliminando almeno una di queste condizioni.

- 📊 L’algoritmo del banchiere aiuta a prevenire e gestire lo stallo verificando se lo stato è sicuro.

📚 ALGORITMO DEL BANCHIERE PIÙ ALTRO (chiesto spesso dal prof)

👨‍💻 1. Algoritmo del banchiere

L’**algoritmo del banchiere** serve per evitare che i processi vadano in **deadlock (blocco totale)** quando richiedono risorse.
👀 **Obiettivo:** **verificare** se **tutti i processi** possono essere completati senza esaurire risorse.

⚙️ **Fasi dell'algoritmo:**

1️⃣ **Inizializzazione:**
✅ `W = V` (dove `W` rappresenta le risorse disponibili).
2️⃣ **Controllo dei processi:**
🔄 Finché esiste un processo `Pi` che può essere completato:
📌 Condizione: `Richiesta di Pi ≤ W` per ogni risorsa `j`

- ✔️ Segna `Pi` come completato
- 🔧 Restituisci le risorse usate da `Pi`:
  ➡️ `W = W + risorse allocate a Pi`
  3️⃣ **Verifica finale:**
- 🟢 **Se tutti i processi sono completati:** lo stato è SICURO.
- 🔴 **Se rimangono processi bloccati:** lo stato è INSICURO.

Chiarimento:

L'espressione **Richiesta di Pi ≤ W** significa:

> "Il processo `Pi` sta chiedendo una quantità di risorse che il sistema ha attualmente disponibili."

Esempio pratico:

- Supponiamo che `W = [5 unità di RAM, 3 unità di CPU]`.
- Il processo `P1` chiede `[3 unità di RAM, 2 unità di CPU]`.

Dal momento che `3 ≤ 5` e `2 ≤ 3`, possiamo dire che la richiesta di `P1` è compatibile con le risorse disponibili (`Richiesta di P1 ≤ W`).
Quindi il sistema può soddisfare la richiesta senza esaurire le risorse.

👨‍💻 2. Algoritmo `safe()` per uno stato sicuro

Questo algoritmo verifica se lo stato del sistema è **sicuro (`true`)** o **insicuro (`false`). Come?** → Simulando l'esecuzione dei processi per **verificare se**, partendo dallo stato S, **NON si verificherà un deadlock**.

🧩 **Struttura dati:**
- 🗂️ **`currentavail`**: array delle risorse disponibili.
- 📋 **`rest`**: lista dei processi ancora da completare.
- ⚡ **`possible`**: booleano che indica se c'è almeno un processo che può essere eseguito in sicurezza

🔧 **Come funziona:**

1️⃣ **Inizializzazione:**
✅ `currentavail = risorse disponibili`
✅ `rest = tutti i processi`
2️⃣ **Simulazione:**
🔄 Finché esiste un processo `Pk` con richieste compatibili (`Richiesta ≤ currentavail`):

- 💡 "Fai finta" di completare `Pk`.
- ➕ Aggiungi risorse usate da `Pk` a `currentavail`.
- ❌ Rimuovi `Pk` da `rest`.

  3️⃣ **Risultato:**
- 🟢 **Se `rest` è vuoto:** stato sicuro (`true`)
- 🔴 **Se `rest` NON è vuoto, cioè restano processi:** stato insicuro (`false`)

🔄 **Ricapitolando il funzionamento:**
Prima di assegnare risorse a un processo, il sistema verifica che la richiesta non superi le risorse totali.

- Se la ***richiesta supera***: il **processo** viene **sospeso** ⏸️
- Se la ***richiesta è valida***: si **simula** l'**allocazione**. Se lo stato risultante è sicuro, la richiesta viene confermata; altrimenti, si ripristina lo stato precedente 🔄

🚨 **3. Come rilevare uno stallo (deadlock) ?**

🔍 **Obiettivo:** controllare periodicamente che tutti i processi possano terminare senza blocchi.

🛠️ **Dati usati:**

- 📊 Matrice delle allocazioni `A(i, j)`
- 📈 Vettore delle risorse disponibili `V`
- 📋 Matrice delle richieste `Q`

🔄 **Come funziona:**

1️⃣ Imposta `W = V`
2️⃣ 🔄 Finché esiste un processo `Pi` con richieste compatibili (`Qi,j ≤ Wj`):

✔️ Completa `Pi`
➕ Restituisci risorse allocate (`W = W + Ai`)
3️⃣ 🔴 **Processi non completati:** sono in **stallo (deadlock)**

♻️ **4. Strategie di ripristino in caso di stallo (deadlock):**

1️⃣ 🛑 **Terminare tutti i processi bloccati:**

- ✅ Soluzione semplice
- ❌ Poco efficiente (i processi potrebbero bloccarsi di nuovo).

2️⃣ 🔄 **Ripristinare i processi in uno stato precedente:**

- 📌 Richiede checkpoint (salvataggi periodici).

3️⃣ ❌ **Terminare un processo alla volta:**

- 🎯 Obiettivo: fermare il minor numero di processi possibile.

4️⃣ 🔧 **Revocare risorse:**

- 🟥 Revoca il **minor numero di risorse possibile**.

📊 **5. Tabella di confronto delle strategie:**

Metodo	Funzionamento	Ciò che gli serve per funzionare	Vantaggi	Svantaggi
✅ Prevenzione	Verifica se ci sono abbastanza risorse prima di avviare un processo	Conoscenza preventiva delle risorse richieste	Evita deadlock	Poco efficiente
🚀 Evitamento	Assegna risorse gradualmente mantenendo il sistema sicuro (cioè è l'allocazione dinamica)	Controllo costante della situazione e possibilità di riprendere risorse	Mantiene stabilità	Rallenta avvio
🔍 Rilevazione	Verifica periodica per individuare blocchi già avvenuti	Meccanismi di monitoraggio dello stato dei processi	Utile se rari errori	Complesso recupero

---

📌 GESTIONE DELLA MEMORIA 📌

❓ Cos'è la **gestione della memoria**?

🖥️ Quando eseguiamo un programma, il sistema operativo assegna una parte della **RAM** per caricare il codice e i dati necessari.
🎯 L'obiettivo è **ottimizzare lo spazio** e **garantire sicurezza** tra i processi.

🛠️ COMPITI DEL SISTEMA OPERATIVO

1️⃣ **Rilocazione** → **Tradurre gli indirizzi** (logici) del programma **in indirizzi fisici** della RAM.
2️⃣ **Protezione** → Impedire che un processo acceda alla memoria di un altro.
3️⃣ **Condivisione** → Permettere a più processi di usare la stessa memoria.
4️⃣ **Organizzazione logica** → Dividere la memoria in moduli (es. librerie).
4️⃣ **Organizzazione fisica** → Gestire come i dati sono distribuiti tra memoria principale e secondaria.

1️⃣ RILOCAMENTO 🔄

📍 I programmi **non sanno** in quale punto della memoria verranno caricati.
📍 Il sistema operativo **traduce** gli indirizzi simbolici in indirizzi fisici reali.
📍 Questo avviene **durante l'esecuzione** e può cambiare più volte.

2️⃣ PROTEZIONE 🛡️

🔒 Un processo **non può accedere** alla memoria di un altro processo.
🔒 Evita errori e attacchi dannosi.
🔒 Controllato da **meccanismi hardware** (registri che verificano gli indirizzi).

3️⃣ CONDIVISIONE 🔄

👥 Alcuni programmi devono **condividere memoria** (es. librerie).
👥 Il sistema operativo garantisce che **solo i processi autorizzati** possano accedere.

4️⃣ ORGANIZZAZIONE DELLA MEMORIA 🏗️

📌 A. Organizzazione Logica

🗂️ Come il programmatore vede la memoria (moduli, librerie, strutture dati).
🔑 Permette di proteggere certe aree (es. librerie di sola lettura).

📌 B. Organizzazione Fisica

💾 Come la memoria è effettivamente gestita dal sistema.
🔄 Se la RAM non è sufficiente, si usano tecniche come **overlaying** (caricare e scaricare parti di codice).

5️⃣ PARTIZIONAMENTO DELLA MEMORIA 🧩

📌 Per caricare più programmi, la memoria è divisa in **partizioni** (blocchi).

🔷 TIPI DI PARTIZIONAMENTO

🟢 **Partizionamento Statico (fisso)**
📏 La memoria è divisa in blocchi di **dimensione fissa**.
⚠️ **Problemi:**
❌ Se il programma è più piccolo, spreca spazio.
❌ Se è più grande, serve **overlaying** (caricare e scaricare parti di codice).

🔵 **Partizionamento Dinamico (variabile)**
📌 Le partizioni vengono create **al momento del caricamento del programma**.
📌 **Metodi per assegnare lo spazio:**
➡️ **Best-Fit** → Si sceglie lo spazio **più vicino** alla dimensione richiesta.
➡️ **First-Fit** → Si prende **il primo spazio libero** abbastanza grande.
➡️ **Next-Fit** → Simile a First-Fit, ma parte **dall'ultima posizione usata**.

6️⃣ FRAMMENTAZIONE DELLA MEMORIA ⚡

🔸 **Frammentazione Interna**

🧩 Quando **una partizione è più grande del necessario** per un programma, viene lasciato parte dello spazio inutilizzato all'interno della partizione stessa.
🚀 **Soluzione alla frammentazione interna:** gestione più efficiente delle dimensioni delle partizioni o l'uso di tecniche di ***allocazione più dinamiche come*** → la **suddivisione della memoria** o la **gestione a pagine**.

🔸 **Frammentazione Esterna**

📌 Quando ci sono **piccoli spazi liberi sparsi tra partizioni occupate**, ma nessuno abbastanza grande per accogliere un nuovo programma.
🚀 **Soluzione alla frammentazione esterna: Compattazione** → spostare i programmi in modo da "raccogliere" questi spazi vuoti e creare aree libere più grandi.

🏁 CONCLUSIONE

💡 La gestione della memoria è fondamentale per l'efficienza del sistema operativo.
📌 Il sistema operativo deve:
✅ Assegnare la memoria ai programmi in modo intelligente.
✅ Proteggere la memoria da accessi non autorizzati.
✅ Ottimizzare lo spazio disponibile per evitare sprechi.

📌 GESTIONE DELLA MEMORIA: BUDDY SYSTEM, PAGINAZIONE E SEGMENTAZIONE. TRADUZIONE DEGLI INDIRIZZI. (chieste spesso dal prof)

1️⃣ Buddy System

📌 Cos'è?

Il Buddy System è un algoritmo di gestione degli spazi liberi in memoria. Divide lo spazio disponibile in blocchi che sono sempre potenze di 2 (es. 2 KB, 4 KB, 8 KB, 16 KB, ...). Questo **algoritmo alloca e dealloca blocchi di memoria** (solitamente per il sistema operativo o per allocazioni dinamiche), NON HA NULLA A CHE FARE CON LA TRADUZIONE DEGLI INDIRIZZI.

📌 Come funziona?

✅ All'inizio, tutta la memoria è un unico grande blocco.

- Supponiamo di avere **256 KB di memoria totale**, e quindi il sistema parte con **un blocco da 256 KB (2^8 KB)** disponibile per l'uso.
- Quando un processo richiede memoria, il sistema cerca il **blocco più piccolo possibile** che sia almeno grande quanto la richiesta.

✅ Suddivisione (divisione in "buddies")

- Se il blocco più piccolo disponibile che è stato trovato è troppo grande, il sistema lo **divide a metà** (e le due metà vengono chiamate **buddies** (compagni)).

📌 Esempio

- Supponiamo di avere **un blocco da 128 KB (2^7 KB)** e un processo richiede **64 KB (2^6 KB)**.
- Il sistema **divide il blocco da 128 KB in due blocchi da 64 KB**.
- Uno dei due blocchi da 64 KB viene assegnato al processo, mentre l'altro rimane libero.

✅ Unione (fusione dei buddy)

- Quando entrambi i buddies sono liberi e vicini, si **uniscono** per formare il blocco originale. Questo aiuta a ridurre la frammentazione della memoria.

📌 Esempio di fusione

- Se due blocchi liberi da **64 KB** (che erano stati ottenuti dividendo un blocco da 128 KB) non sono più usati, vengono riuniti per tornare a formare **un unico blocco da 128 KB**.

✔️ Vantaggi:

- Riduce **solo la frammentazione esterna.**
- Blocchi possono essere riuniti quando diventano liberi.

❌ Svantaggi:

- Può creare **frammentazione interna** perché i blocchi sono sempre di potenze di 2 (**es. una richiesta di 34 KB** userà un blocco di 64 KB (perché 2^5 = 32 quindi non basta perché 34 è più grande di 32 e quindi passa a quello dopo cioè 2^6 = 64), **sprecando 30 KB**, quindi se il blocco assegnato è più grande del necessario, si spreca memoria).

2️⃣ Tipi di Indirizzi in Memoria

🔹 **Indirizzo Logico (o Virtuale)** → indirizzo generato dal programma, indipendente dalla posizione fisica nella RAM.
👉 Quando un programma usa una variabile, usa un indirizzo logico.
🔹 **Indirizzo Fisico** → indica dove si trova il dato nella RAM.
🔹 **Indirizzo Relativo** → È un indirizzo calcolato in base a un riferimento, come la posizione iniziale del programma.

📌 Conversione tra Indirizzi

Quando un programma deve accedere alla RAM che succede? Visto che il programma lavora con indirizzi logici e visto che esso viene eseguito dalla CPU (essa legge e processa il programma (che è un insieme di istruzioni e dati)), **è la CPU che va a leggere e scrivere nella RAM**, ma come fa? **La CPU manda un indirizzo logico** che viene convertito dalla MMU (Memory Management Unit), la **MMU converte quell'indirizzo logico nel corrispondente indirizzo fisico, così la CPU sa dove cercare il dato (che vuole leggere e scrivere)**.

📌 Tipi di Allocazione:

Quando un programma viene caricato in memoria (dal sistema operativo (OS)) ed eseguito (dalla CPU), diventa un **processo**. A questo punto, la memoria può essere gestita in due modi diversi:
✔️ **Allocazione Statica** → Il **processo** viene caricato in una **posizione fissa della RAM** e non si sposta mai durante l'esecuzione.
Un processo viene caricato all'indirizzo fisico `0x1000` e rimane lì fino alla terminazione. La MMU mappa (qua è una semplice conversione) gli indirizzi logici del processo direttamente a `0x1000 + offset`.
✔️ **Allocazione Dinamica** → Il **processo** può essere **spostato in diverse aree della RAM** durante l'esecuzione (richiede hardware speciale per la gestione). Es. il processo viene spostato da `0x1000` a `0x2000`, qui la MMU non si limita a convertire, ma **mantiene una mappa** (tabella) degli indirizzi logici → fisici.

- Questa mappa può cambiare nel tempo: se l'OS sposta il processo in RAM (es. da `0x1000` a `0x2000`), la MMU **aggiorna la sua mappa** (es. tabelle delle pagine o dei segmenti).
- Esempio:
  Prima: Indirizzo logico `0x50` → fisico `0x1000 + 0x50 = 0x1050`.
  Dopo lo spostamento: Indirizzo logico `0x50` → fisico `0x2000 + 0x50 = 0x2050`.

💡 **Collegamento**: La conversione tra indirizzi (logici → fisici) avviene sempre, sia che il processo abbia un'allocazione statica o dinamica. Tuttavia, con l'allocazione dinamica, il processo può essere spostato in RAM, quindi la MMU deve aggiornare la mappatura tra indirizzi logici e fisici ogni volta che cambia posizione.

Chiarimento:

1. **Il programma è un file su disco (memoria di massa)** → Prima di essere eseguito, è solo un insieme di istruzioni salvate su un file (es. un file **.exe** su Windows o un eseguibile su Linux).

2. **Il sistema operativo carica il programma in memoria** → Non è la CPU che lo carica direttamente, ma il **sistema operativo (OS)**. Quando avvii un programma, l'OS riserva spazio in RAM e copia il codice e i dati del programma nella memoria.
3. **Diventa un processo** → Una volta nella RAM, il programma viene gestito dal sistema operativo come un **processo.**
4. **La CPU esegue il processo** → La CPU legge le istruzioni del processo dalla RAM e le esegue, traducendo gli indirizzi logici in fisici con l'aiuto della **MMU**.

3️⃣ TECNICHE DI GESTIONE DELLA MEMORIA:

🔹 PAGINAZIONE (Paging)

📌 **Cos'è?**
La paginazione è una tecnica di gestione della memoria RAM.
Divide la memoria fisica in **blocchi fissi** chiamati **frame** (es. 4KB, 8KB ecc.) e divide il programma in unità della stessa dimensione dei frame chiamate **pagine (blocchi logici dell'indirizzo virtuale del programma)**.
👉 **Esempio:**
Un programma di **12 KB**, diviso in pagine da **4 KB**, avrà **3 pagine logiche** (**12 KB / 4 KB = 3**).
Queste pagine possono essere caricate in **3 frame fisici non per forza consecutive** in RAM.

📌 **Caratteristiche:**
✅ I frame hanno tutti la **stessa dimensione (dimensione fissa)**.
✅ Le pagine del programma possono essere caricate in **qualsiasi frame libero** (non devono essere consecutive).
✅ Il sistema usa una **tabella delle pagine** per tradurre gli indirizzi logici in fisici (mappa ogni pagina logica a un frame fisico).

✅ Ogni pagina è identificata da un **indirizzo logico** che è diviso in **due parti:** un **numero di pagina** (indica quale pagina del programma vogliamo accedere) e **offset** (indica la posizione esatta all'interno della pagina).
✅ Ogni pagina può avere **permessi specifici**, ciò porta **maggiore protezione**.

📌 **Vantaggi:**
✔️ Ottimizza l'uso della memoria.
✔️ Facilita lo **swapping** (spostamento delle pagine tra RAM e disco (memoria virtuale)).
✔️ **Maggiore protezione** perché ogni pagina ha permessi specifici.

📌 **Svantaggi:**
❌ Può causare **frammentazione interna** (alcuni frame potrebbero rimanere inutilizzati (spreco di memoria). **Esempio**: Un processo di 5KB con pagine da 4KB occupa 2 frame (uno da 4KB e un altro da 4KB), spreco = 3KB.
❌ La **tabella delle pagine occupa memoria e richiede accessi aggiuntivi** (risolto con la **TLB**, Translation Lookaside Buffer).

Chiarimento:

Nel contesto della **paginazione**, la **memoria virtuale** è un concetto che permette di trattare la memoria come se fosse più grande di quella fisicamente disponibile nella **RAM**. In realtà, quando la memoria RAM si riempie, alcune pagine possono essere **"spostate"** (o **swapate**) dalla RAM al **disco rigido** o **SSD**, che funge da **memoria virtuale**. Questo processo è chiamato **swapping** e consente di liberare spazio nella RAM per caricare altre pagine, migliorando l'efficienza dell'uso della memoria.

🔹 **SEGMENTAZIONE**

📌 **Cos'è?**
La segmentazione è una tecnica di gestione della memoria RAM.

Divide un programma in **segmenti (blocchi logici,** es. segmento codice (istruzioni), segmento dati (variabili globali), stack (chiamate di funzione)) ognuno con **dimensione variabile**.

📌 **Caratteristiche:**
✅ Ogni segmento ha una **dimensione variabile** cioè legata alle **esigenze del programma** (es. il segmento dati può essere più grande se ci sono molte variabili).
✅ Ogni segmento è identificato da un **indirizzo logico** che è **diviso in due parti:** un **numero di segmento** (indica quale segmento vogliamo usare (es. codice, dati, stack) e un **offset** (spostamento interno, indica la posizione all'interno di quel segmento).
✅ Ogni segmento ha il proprio **indirizzo base** (indirizzo di partenza del segmento in memoria fisica) e un **limite** (lunghezza massima).
✅ Ogni segmento può avere **permessi specifici** (o flag)(es. codice = solo lettura/esecuzione, stack = lettura/scrittura), ciò porta **maggiore protezione**.

📌 **Vantaggi:**
✔️ Modella meglio la **struttura logica** di un programma, separando codice, dati e stack.
✔️ Permette un **uso più flessibile della memoria** rispetto alla paginazione perché i segmenti come il codice possono essere condivisi tra processi (es. librerie condivise).
✔️ **Maggiore protezione** perché ogni segmento ha permessi specifici.

📌 **Svantaggi:**
❌ Può causare **frammentazione esterna** (spazi liberi tra segmenti che non possono essere utilizzati efficientemente). Esempio: Se ho segmenti di 100KB, 200KB e 50KB liberi sparsi, non puoi allocare un segmento da 300KB.
❌ L'allocazione dinamica richiede **algoritmi complessi per trovare spazi adatti** (es. first-fit, best-fit, worst-fit).
❌ Serve una **tabella dei segmenti** (segment table) per mappare segmenti logici a indirizzi fisici, con **costi aggiuntivi** in termini di memoria e tempo.

4️⃣ **Traduzione degli Indirizzi**

📌 **Come funziona?**
🔹 1️⃣ **Nella Paginazione** ✅

- L'**indirizzo logico** è diviso in **Numero di Pagina + Offset**.
- La Tabella delle Pagine traduce il numero di pagina in un **Frame fisico**.
- L'**indirizzo fisico** finale è **Frame fisico + Offset**.

🔹 2️⃣ **Nella Segmentazione** ✅

- L'**indirizzo logico** è diviso in **Numero di Segmento + Offset**.
- La Tabella dei Segmenti traduce il numero di segmento in una **Base fisica**.
- L'indirizzo fisico finale è **Base fisica + Offset**.

🔹 3️⃣ **Nella Paginazione con Segmentazione** ✅

- Unisce le due tecniche: ogni **segmento** è diviso in **pagine**.
- **Prima** si trova il **segmento**, **poi** la **pagina dentro** quel segmento, e **infine** si calcola l'**indirizzo fisico**.

🔹 4️⃣ **Nella RAM (Memoria Continua che viene Spostata Dinamicamente)** ✅

- Anche senza paginazione/segmentazione, può esserci una traduzione degli indirizzi.
- Il sistema usa un **Registro Base per tradurre** l'indirizzo **logico in** indirizzo **fisico**.
- C'è anche un **Registro Limite** che **controlla** che l'**indirizzo non superi i limiti di memoria RAM** assegnati al processo.

  ✅ **Se l'indirizzo è valido** → Si accede alla memoria.
  ❌ **Se l'indirizzo è fuori dai limiti** → Errore, il programma non può accedere.

Chiarimento:
✅ La traduzione dell'indirizzo logico avviene SOLO QUANDO IL PROCESSO DEVE ACCEDERE ALLA MEMORIA RAM.

✅ La RAM è la memoria fisica effettiva su cui il Sistema Operativo (OS) gestisce l'esecuzione dei processi.

📌 RIASSUNTO VELOCE

- **Buddy System:** Divide la memoria in blocchi di potenze di 2, che possono dividersi e riunirsi.
- **Indirizzi:** Logico (dal programma), Fisico (nella RAM), Relativo (rispetto a un altro indirizzo).
- **Allocazione:** Statica (fissa) o Dinamica (può variare).
- **Paginazione:** Divide memoria in blocchi fissi (frame), usa tabelle delle pagine.
- **Segmentazione:** Divide memoria in segmenti logici (codice, dati, stack).
- **Protezione nella RAM:** La RAM usa registri base e limite per impedire accessi non autorizzati o per impedire il tentativo di un processo di accedere a porzioni di memoria che non gli sono assegnate.

🔹 La **paginazione evita la frammentazione esterna**.
🔹 La **segmentazione rispecchia meglio la struttura del programma**.
🔹 Il **Buddy System** è un **compromesso** tra partizionamento fisso e dinamico.

📌 GESTIONE DELLA MEMORIA: PAGINAZIONE E SEGMENTAZIONE (ripetizione delle cose dette prima)

🖥️ ESEMPIO DI GESTIONE DELLA MEMORIA

📌 Immaginiamo di avere **4 processi** e che il primo occupi **4 pagine** (quindi usa 4 frame in memoria).
📍 Il **Processo A** viene caricato e riceve le **pagine 0 e 3**.
📍 Poi vengono caricati gli altri processi.

🆕 Ora arriva un processo che ha bisogno di **2700 byte di memoria**.
🔢 Se un'istruzione si trova al **byte 1502** (indirizzo logico), dobbiamo convertirlo in **indirizzo fisico** per accedere alla memoria.

✅ **Se il processo è stato caricato in una partizione della memoria RAM** → Basta sommare **1502 all'indirizzo della partizione** per ottenere l'indirizzo fisico.
⚠️ **Prima di usare l'indirizzo, dobbiamo verificare** che non superi il limite della memoria assegnata.

📌 PAGINAZIONE 🧩

📝 La **paginazione** suddivide il processo in **pagine di dimensione fissa**.

⚠️ Problema → **Frammentazione interna** (spazio sprecato se la pagina non è riempita completamente).
📍 L'indirizzo logico rimane sempre lo stesso, ma deve essere **convertito** in indirizzo fisico.

🔢 CONVERSIONE DELL'INDIRIZZO NELLA PAGINAZIONE

1️⃣ **Dividiamo l'indirizzo logico per la dimensione della pagina**:

- Se la **pagina è di 1024 byte** e l'indirizzo è **1024 + 478**, significa che siamo nella **pagina 1** (1024 è la prima pagina, 478 è l'offset).
  2️⃣ **Otteniamo la posizione nella pagina**:
- Indice della pagina: **000001 (pagina 1)**
- Offset (posizione relativa dentro la pagina): **478**
  3️⃣ **Consultiamo la tabella delle pagine** per sapere dove si trova la pagina 1 in memoria.

📌 **In generale**:
➡️ **I primi bit dell'indirizzo logico identificano la pagina**.
➡️ **Gli ultimi bit sono l'offset dentro quella pagina**.
➡️ **La tabella delle pagine** contiene la posizione effettiva in memoria di ogni pagina del processo.
➡️ **Alla fine otteniamo l'indirizzo fisico**, che viene usato per accedere alla memoria.

📊 VANTAGGI E SVANTAGGI DELLA PAGINAZIONE

✅ **Evita la frammentazione esterna** → Blocchi di dimensione fissa evitano sprechi tra processi.
❌ **Può causare frammentazione interna** → Se una pagina non è completamente riempita, lo spazio viene sprecato.

📌 SEGMENTAZIONE 🏗️

📝 La **segmentazione** suddivide la memoria in **segmenti di lunghezza variabile** (es. codice, dati, stack).

🔢 CONVERSIONE DELL'INDIRIZZO NELLA SEGMENTAZIONE

1️⃣ **L'indirizzo logico ha**:

- **12 bit per l'offset**
- **4 bit per il segmento**
  2️⃣ **Si cerca nella tabella dei segmenti** l'indirizzo base e la lunghezza del segmento.
  3️⃣ **Si somma l'offset all'indirizzo base** per ottenere l'indirizzo fisico.
  4️⃣ **Si verifica che l'indirizzo ottenuto non superi la lunghezza del segmento** → altrimenti si genera un errore.

📊 VANTAGGI E SVANTAGGI DELLA SEGMENTAZIONE

✅ **Più flessibile della paginazione** → I segmenti sono della dimensione giusta per ogni programma.
❌ **Può causare frammentazione esterna** → I segmenti hanno dimensioni diverse e lasciano spazi vuoti in memoria.

📌 CONCLUSIONE 🏁

🧩 **Paginazione → Divide la memoria in blocchi fissi** → Previene frammentazione esterna ma può sprecare spazio.
🏗️ **Segmentazione → Divide la memoria in blocchi variabili** → Più flessibile ma può creare frammentazione esterna.
📌 **Entrambi i sistemi usano una tabella** per tradurre gli **indirizzi logici in fisici**.
🚀 **Questi metodi migliorano l'uso della memoria e ottimizzano l'efficienza del sistema operativo!**

📌 MEMORIA VIRTUALE 🖥️💾 (chiesta spesso dal prof)

✅ **Cos'è la Memoria Virtuale?**
📍 La **memoria virtuale** è una tecnica che permette di eseguire programmi **più grandi della RAM (memoria fisica disponibile)**.

📍 Funziona **simulando uno spazio di memoria più grande** di quello realmente presente, combinando RAM e memoria di massa (disco).

✅ Come funziona la memoria virtuale?

La memoria virtuale non è un componente fisico, ma un meccanismo che **sfrutta** sia la **RAM** (veloce, ma limitata) **che** il **disco** (lento, ma capiente) **per gestire la memoria** in modo efficiente.

1️⃣ Struttura della memoria (qua per memoria intendo memoria = RAM + disco):

- **RAM (Memoria Fisica)**:
  - 🔹 Divisa in **frame** (**blocchi fissi**, es. 4KB).
  - 🔹 **Contiene le pagine** (ovvero dei **blocchi logici** dell'indirizzo virtuale del programma) attualmente **in uso**.
- **Disco (Memoria di Massa)**:
  - 🔹 Contiene le **pagine** (ovvero **blocchi logici** dell'indirizzo virtuale del programma) **non attive**.
- **Tabella delle pagine**:
  🗂️ Mappa del sistema operativo che collega ogni pagina al suo frame (se in RAM) o alla sua posizione su disco.

2️⃣ Caricamento delle pagine e page fault

- All'avvio di un programma, **solo alcune pagine** vengono caricate in RAM (es. codice iniziale, dati essenziali).
- Se il **programma richiede una pagina non presente in RAM** → si verifica un **page fault** 🚨
  📌 Il sistema operativo:
  1. Sospende il programma.
  2. Trova spazio in RAM (rimuovendo pagine non usate, se necessario).
  3. Carica la pagina richiesta dal disco alla RAM.
  4. Riprende l'esecuzione.

3️⃣ Cosa succede durante un Page Fault 🚨?

Ecco lo scenario:

1. Il programma cerca di accedere a una pagina **X**.
2. La CPU controlla la **tabella delle pagine** (una mappa che dice dove sono le pagine in RAM).

3. Se la pagina **X non è in RAM → Page Fault**!

Cosa fa il sistema operativo:

- **Step 1**: Sospende temporaneamente il programma (non viene chiuso!).
- **Step 2**: Trova spazio in RAM (se serve, elimina una pagina non usata).
- **Step 3**: Carica la pagina **X** dal disco (HDD/SSD) alla RAM.
- **Step 4**: Aggiorna la tabella delle pagine: "La pagina X ora è in RAM!".
- **Step 5**: Riprende il programma come se nulla fosse.

4️⃣ Principio di Località

- I programmi tendono ad accedere a:
  - 🔹 **Dati/codice usati di recente** (località temporale).
  - 🔹 **Dati/codice vicini in memoria** (località spaziale).
- 📉 Questo **riduce i page fault** e migliora le prestazioni.

👉 **Esempio**:
Se stai usando una funzione di Photoshop, è probabile che userai ancora quella funzione o strumenti vicini.

❓ Perché non tutte le pagine vengono caricate subito in RAM?

- 1️⃣ **RAM limitata**: Non è fisicamente possibile caricare programmi molto grandi (es. 10GB) in una RAM da 8GB.
- 2️⃣ **Efficienza**: Molte parti di un programma (es. funzioni rare o dati non usati) potrebbero non servire subito.

👉 **Esempio**: Immagina un videogioco. All'inizio carica solo i menu, non i livelli avanzati che userai dopo ore!

❓ Perché è utile la memoria virtuale?

- **Efficienza**: Non sprechi RAM con dati inutili.
- **Trasparenza**: Il programma "vede" tutta la memoria (virtuale), anche se fisicamente non esiste!
- **Multitasking**: Più programmi possono convivere in RAM senza sovrapporsi.

❗ Attenzione con la memoria virtuale perché ci possono essere:

- 🚨**Page fault frequenti** = Problema! Significa che il sistema passa più tempo a caricare pagine che a eseguire il programma (si chiama **thrashing**).

- 🚨 **Overhead della tabella delle pagine → Gestire le pagine richiede risorse aggiuntive (tempo, memoria, energia)** che un sistema consuma per gestire un processo.
- Soluzione per i page fault frequenti: Aggiungere più RAM o ottimizzare il programma.
- Soluzione per l'overhead: TLB (Translation Lookaside Buffer) o paginazione gerarchica o ottimizzazione delle dimensione delle pagine.

📌 Traduzione degli indirizzi nella paginazione 🔄

Un programma usa indirizzi logici (virtuali), ma la RAM lavora con indirizzi fisici.
Il sistema converte un indirizzo logico in un indirizzo fisico utilizzando la **tabella delle pagine**.

🧮 **Formula**:
🔢 **Indirizzo Logico = Numero di Pagina + Offset**

- 📌 **Numero di Pagina** → Indica **quale pagina** serve.
- 📌 **Offset** → Indica la **posizione dentro la pagina**.

✅ Il sistema consulta la **tabella delle pagine**, trova il numero di frame corrispondente e ottiene l'**indirizzo fisico**.

📌 Sostituzione pagine (page replacement) 🔄

Se la **RAM è piena** e serve una nuova pagina, il sistema deve **rimuovere una pagina esistente** per fare spazio.
📌 **Algoritmi comuni per scegliere quale pagina rimuovere**:

1️⃣ **FIFO (First In, First Out)** 🔁 → Si rimuove la **pagina più vecchia**.
2️⃣ **LRU (Least Recently Used)** ⏳ → Si rimuove la **pagina meno usata di recente**.
3️⃣ **Optimal (Ottimale)** 🎯 → Si rimuove la **pagina che non servirà per più tempo**.

🔍 PAGINAZIONE (PAGING) VS. SEGMENTAZIONE

📂 Paginazione	📁 Segmentazione
📌 Memoria divisa in **pagine** di dimensione **fissa**.	📌 Memoria divisa in **segmenti** di dimensione **variabile**.
📌 Il **sistema operativo** gestisce le pagine.	📌 Il **programmatore** gestisce i segmenti.
✅ **Evita problemi di frammentazione esterna**.	❌ **Può causare frammentazione esterna**.
📌 **Più adatto ai sistemi operativi moderni**.	📌 Utile quando i dati hanno **strutture logiche separate**.

📌 RIASSUNTO 🏁

✅ La memoria virtuale permette di eseguire programmi più grandi della RAM disponibile.
✅ Le pagine vengono caricate in memoria solo quando servono.
✅ Gli indirizzi logici vengono tradotti in indirizzi fisici tramite la tabella delle pagine.
✅ Se la RAM è piena, il sistema deve sostituire una pagina.
✅ Paginazione e segmentazione sono due tecniche diverse per la gestione della memoria.

🚀 Questi metodi migliorano l'uso della memoria e ottimizzano l'efficienza del sistema operativo!

📌 TABELLA DELLE PAGINE A DUE LIVELLI E TABELLA DELLE PAGINE INVERTITA🗂️

📌 TABELLA DELLE PAGINE A DUE LIVELLI

⚠️ Problema

📍 Se la **tabella delle pagine è troppo grande**, non può essere contenuta in una sola pagina di memoria.
📍 Serve un **modo più efficiente per dividerla e gestirla**.

✅ Soluzione

📌 Si usa una **Tabella delle Pagine a Due Livelli**, con una **tabella radice** che punta a **sotto-tabelle delle pagine**.
📌 Questo sistema aiuta a **suddividere** la gestione degli indirizzi logici.

📌 Come funziona la tabella delle pagine a due livelli?

1️⃣ **Un indirizzo virtuale è diviso in tre parti**:

- 🔢 **Primi 10 bit** → Indicano **quale sotto-tabella** della pagina deve essere usata.
- 🔢 **Successivi 10 bit** → Indicano **l'elemento specifico** nella sotto-tabella.
- 🔢 **Ultimi bit (offset)** → Indicano **la posizione esatta dentro il frame** in memoria.

2️⃣ Il **sistema operativo** usa la **tabella radice** per trovare la giusta **sotto-tabella**.
3️⃣ La **sotto-tabella** fornisce il **numero di frame** in RAM.
4️⃣ Il valore di **offset** viene aggiunto per **trovare la posizione esatta** in memoria.

📌 Vantaggi 🚀

✔️ **Riduce lo spazio** richiesto per la tabella delle pagine.
✔️ **Permette una gestione più efficiente** della memoria virtuale.

📌 TABELLA DELLE PAGINE INVERTITA 🔄

⚠️ Problema

📍 Con **molti processi attivi**, la **tabella delle pagine diventa enorme** e difficile da gestire.

✅ Soluzione

📌 Si usa una **Tabella delle Pagine Invertita**, che invece di salvare una voce per **ogni pagina del processo**, salva **solo le mappature per le pagine effettivamente caricate** in memoria.

📌 Come funziona la tabella delle pagine invertita?

1️⃣ **L'indirizzo logico viene elaborato con una funzione di hashing** 🔢, che fornisce un **indice** nella tabella.
2️⃣ Se il **numero di pagina** corrisponde a quello cercato, il sistema ottiene il **frame** in memoria.
3️⃣ Se il numero di pagina **non corrisponde**, si passa all'elemento successivo della tabella **fino a trovare il valore giusto**.
4️⃣ Una volta trovato, si utilizza l'**offset** per **accedere alla posizione corretta** nel frame.

📌 Vantaggi 🚀

✔️ **Riduce lo spazio** della tabella delle pagine.
✔️ **Accelera la ricerca** delle pagine in memoria (**in media 1 accesso**).
✔️ **Evita di dover memorizzare** ogni singola pagina di ogni processo.

📌 DIFFERENZE TRA I DUE METODI 🔍

🗂️ Tabella delle Pagine a Due Livelli	🔄 Tabella delle Pagine Invertita
📌 **Divide** la tabella delle pagine in più livelli.	📌 Usa una **funzione di hashing** per ridurre la tabella.
📌 **Ogni processo ha una propria tabella** delle pagine.	📌 Esiste **una sola tabella per tutto il sistema**.
📌 **Buona per gestire grandi spazi di indirizzamento**.	📌 **Buona per ridurre** la memoria occupata dalla tabella.
❌ **Richiede due accessi** alla memoria per trovare un frame.	✅ **Di solito basta un solo accesso**.

📌 RIASSUNTO 🏁

✅ **La Tabella delle Pagine a Due Livelli** suddivide la gestione della memoria per **ridurre la dimensione** della tabella.
✅ **La Tabella delle Pagine Invertita** usa una funzione di **hashing per velocizzare la ricerca** delle pagine e ridurre lo spazio occupato.
✅ **Entrambi i metodi migliorano l'uso della memoria virtuale** nei sistemi operativi.

🚀 Queste tecniche ottimizzano la gestione della memoria e migliorano l’efficienza del sistema!

📌 TLB (TRANSLATION LOOKASIDE BUFFER) ⚡

🧠 Cos'è il TLB?

📍 Il **TLB** è una **memoria cache speciale** che **accelera** la traduzione degli indirizzi virtuali (o logici) in fisici.
📍 Si basa sul **principio di località**: i programmi accedono **spesso** alle stesse pagine.
📍 Contiene le **mappature delle pagine usate di recente**, evitando di consultare sempre la tabella delle pagine.

📌 Come funziona il TLB? ⚙️

1️⃣ **Il processore controlla il TLB** per vedere se l’indirizzo della pagina è **già memorizzato**.
2️⃣ **Se lo trova (✅ TLB hit)** → La traduzione è **veloce** e si ottiene subito l’indirizzo fisico.
3️⃣ **Se non lo trova (❌ TLB miss)** →

- 🔍 Il sistema cerca l’indirizzo nella **tabella delle pagine**.
- 🔄 **Aggiorna il TLB** con la nuova mappatura.
- 🔁 **Ripete l’operazione** per la prossima volta.

📌 ESEMPIO DI TRADUZIONE CON IL TLB 🔄

🖥️ **Supponiamo che la CPU voglia accedere alla pagina 5**:
✔️ **Controlla il TLB** → Se la pagina è presente, trova subito il **numero di frame**.
❌ **Se la pagina non è nel TLB** → Cerca nella **tabella delle pagine** e **aggiorna il TLB** con la nuova mappatura.
🔄 **Se necessario**, il TLB **sostituisce una vecchia mappatura** con la nuova.

📌 Rilocamento con il TLB 🔁

📍 Se una pagina **non è in RAM**, il sistema deve:
1️⃣ Cercarla nella **tabella delle pagine**.
2️⃣ **Caricare la pagina** dalla memoria di massa in RAM.
3️⃣ **Aggiornare la mappatura presente nel TLB**, associando l’indirizzo virtuale già noto al nuovo indirizzo fisico della pagina in RAM.
4️⃣ **Riprendere** l'esecuzione del programma.

📌 GESTIONE DEI PAGE FAULT CON IL TLB ❗

📍 Se una pagina **non è in memoria**, si verifica un **Page Fault** e il sistema operativo interviene:

1️⃣ **Controlla se c'è spazio in RAM**:

- ✅ **Se sì**, carica la pagina direttamente.
- ❌ **Se no**, usa un **algoritmo di sostituzione** (FIFO, LRU…) per rimuovere una pagina meno usata.

2️⃣ **Aggiorna la tabella delle pagine e il TLB**.
3️⃣ **Riprende l'esecuzione della CPU** con il nuovo indirizzo tradotto (ovvero l'indirizzo fisico).

📌 RIASSUNTO 🏁

✔️ Il **TLB accelera** la traduzione degli indirizzi virtuali in fisici.
✔️ Se la pagina è **nel TLB** → La traduzione è **molto veloce** (**TLB hit** ✅).
✔️ Se la pagina **non è nel TLB** → Bisogna cercarla nella **tabella delle pagine** (**TLB miss** ❌).
✔️ Se la pagina **non è in RAM** → Il sistema deve **caricarla dal disco** (**Page Fault** ⚠️).
✔️ Il **TLB viene aggiornato regolarmente** per mantenere **le pagine più usate** in cache.

🚀 Utilizzare il TLB migliora enormemente le prestazioni del sistema! 🎯

📚 PAGINAZIONE, SEGMENTAZIONE, SEGMENTAZIONE PAGINATA ALTRE INFORMAZIONI (chieste spesso dal prof)

🟢 1. Dimensione delle Pagine

📍 **Pagine piccole**
✅ Meno spreco di spazio (**minore frammentazione interna**).
❌ Più pagine → **Tabella delle pagine più grande**.

📍 **Pagine grandi**
✅ Meno pagine → **Tabella delle pagine più piccola**.
✅ **Migliore uso della TLB** (cache per la memoria virtuale).
❌ **Più spreco di spazio** se le pagine non sono usate completamente.

📍 **Scelta della dimensione**
🔹 Dipende dall'**hardware** e dalla **memoria secondaria**.
🔹 Se la pagina è **troppo piccola** → **Troppi cambi di pagina** (**Page Faults** 🚨).
🔹 Se la pagina è **troppo grande** → **Più spazio inutilizzato**.
🔹 **Soluzione**: trovare una **dimensione ottimale** per il programma.

🔵 **2. Vantaggi della Segmentazione** (➡️ Un programma viene diviso in moduli logici chiamati segmenti (es. codice, dati, stack))
✅ Possiamo caricare solo i segmenti necessari, **risparmiando memoria**.
✅ Ogni segmento è indipendente → Più **facile condividere dati tra processi**.
✅ **Più sicurezza**: un segmento non può accedere a un altro per errore.
✅ **Meno cambi di segmento** rispetto ai cambi di pagina → **Più efficienza**.

🟠 **3. Tabella dei Segmenti**

🔹 Funziona in modo diverso dalla **tabella delle pagine**:
📌 Ogni segmento ha:
🔸 **Indirizzo di base** (o iniziale, cioè dove si trova in memoria).
🔸 **Lunghezza massima** (quanti byte occupa).
🔸 **Flag** (es. **lettura/scrittura**, **protezione**).

🔴 **4. Rilocazione con Segmentazione**

📌 **Come funziona?**
🛠️ La **tabella dei segmenti** è **in memoria** e c'è un **registro speciale** che la punta.

🔢 **Passaggi per calcolare l'indirizzo fisico**:
1️⃣ Si prende l'**indice del segmento**.
2️⃣ Si controlla nella **tabella dei segmenti**.
3️⃣ Si ottiene l'**indirizzo base del segmento**.
4️⃣ Si somma l'**offset** all'**indirizzo base**.
5️⃣ Si verifica che l'**offset** non superi la **lunghezza del segmento**.

🛑 **Se l'offset è troppo grande → ❌ Errore di protezione!**

🟡 **5. Segmentazione Paginata (Paginazione + Segmentazione)**

📌 **Cos'è?**
➡️ Combina i vantaggi della **paginazione** e della **segmentazione**.
➡️ **Ogni segmento è diviso in pagine**.

📌 **Struttura dell'indirizzo logico**
🔹 Numero di **segmento**
🔹 Numero di **pagina** dentro il segmento
🔹 **Offset** nella pagina

📌 **Passaggi per tradurre un indirizzo logico in un indirizzo fisico(cioè la posizione effettiva in RAM)**
1️⃣ **Numero di segmento** → Usiamo questo valore per guardare la **tabella dei segmenti** e trovare l'inizio del segmento (cioè il suo indirizzo base in memoria).
2️⃣ **Numero di pagina dentro il segmento** → All'interno del segmento, questo numero viene usato

per cercare, nella relativa tabella delle pagine, la pagina specifica e quindi il suo frame fisico (cioè l'indirizzo base della pagina in RAM).
3️⃣ **Offset dentro la pagina** → Una volta **individuato il frame fisico corrispondente alla pagina**, **l'offset viene aggiunto** a questo indirizzo base del frame per **calcolare l'indirizzo fisico finale** in RAM.

📌 ✅ **Vantaggi**
✔️ **Meno spreco di spazio** rispetto alla segmentazione classica.
✔️ Più flessibilità e **meno frammentazione interna**.
✔️ Ogni segmento può avere pagine di dimensioni diverse (ciò significa che il sistema può scegliere una dimensione di pagina che si adatta meglio alle esigenze specifiche di ciascun segmento) = ottimizzazione dello spazio, adattibilità alle esigenze specifiche, migliore uso della RAM.

💡 **In breve:**
📌 **Paginazione** → Divide il programma in **pagine**.
📌 **Segmentazione** → Divide il programma in **segmenti**.
📌 **Segmentazione Paginata** → Divide i **segmenti in pagine**, combinando i due metodi.

🚀 Conoscere queste tecniche aiuta a migliorare l'uso della RAM nei sistemi operativi! 💡

📚 MEMORIA VIRTUALE ALTRE INFORMAZIONI (chiesta spesso dal prof)

🔹 **Nell'usare la Memoria Virtuale** (tecnica che permette di eseguire programmi più grandi della memoria fisica disponibile) bisogna prendere **4 decisioni importanti:**
1️⃣ **Quale tecnica usare?** (Paginazione, Segmentazione, Segmentazione Paginata).
2️⃣ **Quando i dati (pagine o segmenti) vengono caricati in RAM?** (Strategie di Caricamento).
3️⃣ **Come posizionare i dati in RAM una volta che sono stati caricati in essa?** (Strategie di Posizionamento).
4️⃣ **Quando e cosa sostituire dalla memoria?** (Strategie di Sostituzione).

2️⃣ Strategie di Caricamento: due strategie

Indicano **quando** le pagine o i segmenti vengono caricati in RAM.

🔸 **1° strategia: Paginazione a Richiesta**

- Le pagine vengono caricate solo **quando il processo le richiede**.
- Se una pagina non è in RAM (memoria principale), si genera un **errore di pagina** (**page fault**), e il sistema la carica dal disco (memoria secondaria).
- ✅ Vantaggi: Usa meno RAM, lascia spazio ad altri programmi.
- ❌ Svantaggi: Rischia di causare molti errori di pagina → rallentamento.

🔸 **2° strategia: Prepaginazione**

- Il sistema carica in anticipo le pagine che **prevede** saranno richieste.
- ✅ Vantaggi: Meno errori di pagina.
- ❌ Svantaggi: Potrebbe caricare pagine inutili, sprecando memoria.

3️⃣ Strategie di Posizionamento

Obiettivo:
Decidere in quale punto della RAM inserire i dati (segmenti o pagine) per usare al meglio lo spazio disponibile.

🔹 **1. Posizionamento nella Segmentazione:**

Quando si usa la segmentazione, ogni segmento (cioè una parte logica del programma, come codice, dati, stack, ecc.) può essere collocato in una zona qualsiasi della RAM. Per decidere **dove** posizionare un segmento, si usano algoritmi che cercano di sfruttare al meglio lo spazio libero:

First Fit:
Il sistema scorre la RAM dall'inizio e assegna il segmento al **primo blocco** libero della RAM che è abbastanza grande da contenerlo.

Best Fit:
Il sistema esamina tutti i blocchi liberi della RAM e sceglie quello **più piccolo** possibile che può ancora contenere il segmento. L'obiettivo è ridurre gli spazi inutilizzati.

Next Fit:
Simile al First Fit, ma invece di cominciare sempre dall'inizio della RAM, il sistema parte dall'ultima posizione in cui è stata fatta un'allocazione, continuando la ricerca da lì.

🔹 **2. Posizionamento nella Paginazione:**

Con la paginazione, i dati sono divisi in unità di dimensioni fisse chiamate pagine. In questo caso:

Le pagine vengono **caricate in qualsiasi frame** disponibile**.** Non serve cercare uno spazio "su misura" perché tutte le pagine hanno la stessa dimensione.

- ✅ Vantaggio: Non ha problemi di frammentazione esterna (cioè non rimangono spazi vuoti tra le pagine che non possono essere usati).
- ❌ Svantaggio: Ha frammentazione interna (se una pagina non viene completamente utilizzata, lo spazio rimanente all'interno di quella pagina resta inutilizzato).

4️⃣ Strategie di Sostituzione:

Indicano **quale pagina rimuovere** dalla memoria quando lo spazio è pieno.

🔹 **Frame Protetti:**

- **alcuni frame** (cioè le unità di memoria fisica in cui sono caricate le pagine) sono "**protetti**" e **non possono essere usati per la sostituzione come** nel caso delle **pagine del kernel** del sistema operativo.
- Esiste un **bit di controllo** che indica se una pagina può essere rimossa o meno.

🔹 **Algoritmi di Sostituzione:**

1° **LRU (Least Recently Used)**

- Rimuove la pagina che **non è stata usata da più tempo**.
- ✅ Vantaggi: Efficiente perché segue la probabilità di riuso.
- ❌ Svantaggi: Difficile da implementare.

2° **FIFO (First In, First Out)**

- Rimuove la pagina più vecchia.
- ✅ Vantaggi: Facile da implementare.
- ❌ Svantaggi: Non tiene conto di quanto la pagina sia ancora utile.

3° **Clock (Strategia dell'Orologio)**

- Variante migliorata di FIFO:
  - Ogni pagina ha un **flag di utilizzo** (0 o 1).
  - Se il flag è **1**, lo **azzera e passa** alla pagina **successiva**.
  - Se il flag è **0**, **rimuove** la pagina.
- ✅ Vantaggi: Meno sprechi rispetto a FIFO.

4° **OPT (Sostituzione Ottimale)**

- Sostituisce la pagina che **non sarà richiesta per più tempo in futuro**.
- ✅ Vantaggi: È la migliore strategia teorica.
- ❌ Svantaggi: Impossibile da applicare realmente (non possiamo prevedere il futuro).

🟡 **Buffer delle Pagine**

- **Cos'è?** Una **memoria temporanea per pagine che sono state rimosse** ma potrebbero essere richieste di nuovo.
- ✅ Se una pagina viene rimossa e poi richiesta di nuovo, la recuperiamo subito senza ricaricarla dal disco.
- ✅ Migliora la velocità di accesso ai dati.

💡 **Riassunto Finale:**

- **Memoria virtuale** = Tecnica per gestire programmi più grandi della RAM.
- **Strategie di caricamento** = Decidono **quando** caricare una pagina/segmento.
- **Strategie di posizionamento** = Decidono **dove** posizionare i dati in memoria.
- **Strategie di sostituzione** = Decidono **quale pagina rimuovere** quando la memoria è piena.

- **Buffer delle pagine** = Aiuta a recuperare pagine rimosse di recente più velocemente.

✅ RESIDENT SET E GESTIONE DELLA MEMORIA

(Ottimizzazione dell'uso della RAM nei sistemi multiprocesso)

📌 1. Introduzione

- **Frame**:
  Blocchi fissi di memoria fisica (es. 4 KB) in cui viene divisa la RAM.
- **Resident Set**:
  Insieme di pagine di un processo attualmente caricate in RAM (frame allocati).
- **Problema chiave**:
  - Ogni processo ha un **numero limitato di frame** (memoria fisica è condivisa).
  - **Molti processi attivi** → meno frame per processo → aumento **page fault** (errori di pagina).
- **Obiettivo**:
  Trovare un equilibrio tra:
  - Allocazione efficiente dei frame.
  - Strategia di sostituzione delle pagine per minimizzare i page fault.

📌 2. Strategie di Allocazione dei Frame

(Come distribuire i frame tra i processi)

A. Allocazione Fissa

- **Descrizione**:
  Ogni processo riceve un **numero predefinito di frame** (es. 10 frame/processo).
- **Vantaggi**:
  ✅ Semplice da gestire.
  ✅ Prevedibile.
- **Svantaggi**:
  ❌ **Under-allocation**: Se i frame sono troppo pochi → page fault frequenti.
  ❌ **Over-allocation**: Se sono troppi → spreco di RAM (altri processi ne hanno bisogno).

B. Allocazione Dinamica (Variabile)

- **Descrizione**:
  I frame vengono **ridistribuiti in base alle esigenze** (monitorando i page fault).

- Processi con **molti page fault** → ricevono **più frame** (**priorità**).
- Processi con **pochi page fault** → **cedono frame**.

- **Vantaggi**:
  ✅ Massimizza l’uso della memoria.
  ✅ Adattabile al carico di lavoro.
- **Svantaggi**:
  ❌ Complessità gestionale (richiede monitoraggio continuo).

📌 3. Strategie di Sostituzione delle Pagine

(Quale pagina rimuovere quando serve spazio?)

A. Ambito Locale

- **Descrizione**:
  Il Sistema Operativo sostituisce **solo pagine del processo** che ha generato il page fault.
- **Vantaggi**:
  ✅ Equità: un processo non può "rubare" frame ad altri.
- **Svantaggi**:
  ❌ Se il processo ha pochi frame → page fault ricorrenti.

B. Ambito Globale

- **Descrizione**:
  Il Sistema Operativo sostituisce **pagine di qualsiasi processo** (libera frame ovunque in RAM).
- **Vantaggi**:
  ✅ Maggiore efficienza (usa tutta la memoria disponibile).
- **Svantaggi**:
  ❌ Processi poco attivi potrebbero perdere pagine utili → rallentamenti.

📌 4. Combinazioni di Strategie

(Allocazione + Sostituzione)

Strategia	Descrizione	Vantaggi	Svantaggi
Allocazione Fissa + Locale	**Frame fissi** ai processi + **sostituzione** di **frame** solo **interne** a quei processi.	✅ Niente conflitti tra processi.	❌ Sistema Rigido: i processi con pochi frame possono avere prestazioni inferiori.
Allocazione Dinamica + Globale	**Frame variabili** ai processi + **sostituzione** di **frame** può avvenire **su tutta** la **RAM**.	✅ Migliora l’uso della RAM.	❌ Rischio di starvation per processi lenti.

Strategia	Descrizione	Vantaggi	Svantaggi
Allocazione Dinamica + Locale	**Frame variabili** ai processi + **sostituzione** di **frame** solo **interne** a quei processi.	⚖️ Buon compromesso tra controllo e flessibilità.	❌ Richiede monitoraggio costante.

📌 5. Working Set: Ottimizzare l'Allocazione

(Adattare i frame alle esigenze reali dei processi)

A. Cos'è il Working Set?

- Insieme di pagine **attivamente usate** da un processo in un intervallo di tempo.
- **Obiettivo**:
  Assegnare al processo un numero di frame ≥ alla dimensione del suo working set → **ridurre page fault**.

B. Gestione del Working Set

1. **Monitoraggio**:
   Traccia i page fault e l'uso delle pagine in un intervallo di tempo (es. ultimi 10 ms).
2. **Aggiustamento dei frame**:
   - Se page fault ↑ → alloca più frame al processo.
   - Se page fault ↓ → riduci i frame (libertà per altri processi).
3. **Parametri chiave**:
   - Durata minima di osservazione.
   - Soglia massima di page fault tollerabili.

C. Variable-Interval Sampled Working Set

- **Approccio dinamico**:
  Adatta l'intervallo di osservazione in base al carico del sistema, l'obiettivo è mantenere in RAM le pagine davvero necessarie al processo.
- **Vantaggi**:
  ✅ Riduce page fault in scenari con carico variabile.
  ✅ Evita allocazioni eccessive.

📌 6. Effetti del Working Set

- **Politica di Carico**:
  Limita il numero di processi in RAM per garantire a ciascuno abbastanza frame.
- **Performance**:
  - ⏱️ Meno tempo speso a gestire page fault → **CPU più efficiente**.

- 🚀 Processi completano le operazioni più velocemente.

📌 Conclusione

- **La RAM è una risorsa critica**:
  La gestione ottimale richiede un mix di strategie di allocazione e sostituzione.
- **Compromesso tra flessibilità e complessità:**
  Se scegliamo un'**allocazione dinamica delle risorse**, il **sistema** diventa **più flessibile** e capace di **adattarsi a cambiamenti**, **ma** allo stesso tempo risulta **più complicato** da gestire. Invece, un'**allocazione fissa** rende il sistema **più semplice**, anche se **meno adattabile** alle situazioni variabili.
- **Compromesso tra efficienza ed equità:**
  Utilizzando una **sostituzione globale**, possiamo ottenere una **maggiore efficienza**, sfruttando al meglio tutte le risorse disponibili. Tuttavia, questo approccio potrebbe **non garantire una distribuzione equa**. Al contrario, una **sostituzione locale** tende a favorire una **distribuzione più giusta**, anche se potrebbe comportare una **minore efficienza** complessiva.
- **Working Set**:
  Strumento chiave per adattare l’allocazione alle esigenze dinamiche dei processi.

🖇️ Glossario

- **Page Fault**: Errore che si verifica quando una pagina richiesta non è in RAM e deve essere caricata dal disco.
- **Starvation**: Situazione in cui un processo non riceve risorse sufficienti (es. frame) per avanzare.
- **Working Set**: Pagine attive usate da un processo in un dato intervallo di tempo.

Chiarimento:

Un **processo** in esecuzione è suddiviso in **pagine** (unità **logiche** di memoria), mentre la **RAM** è suddivisa in **frame** (unità **fisiche** di memoria).

Quando un processo deve essere caricato in RAM, le sue pagine vengono assegnate ai frame disponibili. Questo processo di associazione tra pagine e frame è chiamato **mappatura**.

SCHEDULING (chiesto spesso dal prof)

📌 Cos’è lo Scheduling?

Lo **scheduling** è il **meccanismo** con cui il **Sistema Operativo** decide **quale processo eseguire** e in **quale ordine**.
Permette di **gestire più processi insieme** e usare il **processore** in modo **efficiente**.

🔹 Lo scheduling su cosa si basa?

Può basarsi su:
✅ Priorità del processo
✅ Tempo di attesa
✅ Tempo di esecuzione
✅ Altri fattori legati all’efficienza

⚙️ Multitasking e Condivisione del Processore

- In un sistema ci sono **più processi** rispetto ai processori disponibili.
- Lo scheduling **assicura che tutti i processi avanzino**, anche se ci sono pochi processori.
- **Obiettivi principali:**
  ✔️ Usare al massimo la CPU
  ✔️ Dare risposte rapide ai processi

📌 Tipi di Scheduling

Lo scheduling è diviso in **tre livelli principali**:

1️⃣ Scheduling a LUNGO TERMINE (Long-term scheduling)

- **Controlla** quanti **processi** possono stare in **memoria RAM**.
- **Decide** quali processi **caricare** dal disco alla RAM.
- Può **sospendere e ripristinare** processi per gestire meglio la memoria.
- Funziona **a bassa frequenza** (non continuamente).

2️⃣ Scheduling a BREVE TERMINE (Short-term scheduling o Dispatcher)

- **Sceglie** quale **processo eseguire subito** tra quelli pronti.
- Lavora **velocemente** e in modo **continuo**.
- **Ottimizza l’uso del processore** per ridurre i tempi di attesa.
- Funziona **ad alta frequenza**.

3️⃣ Scheduling di I/O (Input/Output scheduling)

- **Organizza l’accesso ai dispositivi** (dischi, stampanti, rete).
- **Riduce** i **tempi di attesa** per le operazioni di input/output.
- **Ottimizza l’uso delle periferiche** (Input/Output) per migliorare le prestazioni.

📌 Politiche di Scheduling

🔹 Non Preemptive

- Un **processo usa** la **CPU fino a quando non finisce** o si blocca.
- Nessun altro processo può interromperlo.

🔹 Preemptive

- Il sistema operativo **può interrompere un processo** e **assegnare** la **CPU a un altro** più urgente.

📌 Algoritmi di Scheduling

Gli algoritmi determinano **l'ordine di esecuzione** dei processi.

1️⃣ First Come First Served (FCFS) – "Chi prima arriva, prima viene servito"

✔️ I processi vengono eseguiti **nell'ordine in cui arrivano**.
❌ Se un processo è molto lungo, gli altri **devono aspettare**. Quindi penalizza i processi brevi perché devono aspettare i processi lunghi (**effetto convoglio**).
✔️ **Minimo overhead** (tempo perso nei cambi di processo).
✔️ Semplice ed equo.

2️⃣ Round Robin (RR) – "Turni di tempo uguali"

✔️ Ogni processo riceve un **tempo massimo di esecuzione (quantum)**.
✔️ Se il processo **non finisce entro il quantum**, viene sospeso e messo in coda quindi **passa il turno** e riprova dopo.
✔️ Garantisce **equità tra i processi**.
✔️ Ottimo per sistemi con più utenti (multiutente).
❌ Se il **quantum è troppo piccolo**, il sistema perde tempo nei cambi di processo (**overhead elevato**).

3️⃣ Shortest Job Next (SJN, chiamato anche SPN (Shortest Process Next)) – "Prima i processi più brevi"

Non preemptive: un processo in esecuzione **non viene interrotto**.
✔️ Si eseguono **prima** i **processi** che richiedono **meno tempo**.
❌ Non sempre è possibile prevedere la durata di un processo.
✔️ **Minimizza** il **tempo** medio di **completamento**.
❌ I processi lunghi possono rimanere bloccati (**starvation**).

3️⃣ SRT (Shortest Remaining Time) → Variante preemptive di SPN – "Prima i processi con il tempo rimanente più breve"

Simile a SPN, ma **preemptive**: se **arriva** un processo più **breve**, **quello** in **esecuzione** viene **interrotto**.
✔️ **Ottimizza** ancora **di più** il **tempo** medio di **completamento**.
❌ **Starvation** ancora più probabile per i processi lunghi.

4️⃣ HRRN (Highest Response Ratio Next) → Equilibrio tra giustizia ed efficienza

Si calcola un punteggio basato su: Tempo di attesa o Tempo richiesto dal processo → Si esegue il processo con il punteggio più alto.
✔️ **Riduce** il rischio di **starvation.**
✔️ **Equilibrio** tra equità ed efficienza.
❌ Più **complesso** da implementare rispetto a FCFS e RR.

5️⃣ Priority Scheduling (Scheduling a Priorità)

✔️ I processi con **priorità più alta** vengono eseguiti **prima**.
❌ I processi a **bassa priorità** potrebbero **non essere mai eseguiti**.

5️⃣ Multilevel Queue – "Code separate per diversi tipi di processi"

✔️ Divide i processi in **gruppi (code) con diverse priorità** (es. processi di sistema vs. processi utente).
✔️ Alcuni di questi gruppi (code) hanno **più priorità** rispetto ad altri gruppi (code).

📌 Obiettivi dello Scheduling

✅ Massimizzare l'uso della CPU
✅ Ridurre i tempi di attesa
✅ Gestire bene la memoria e le periferiche
✅ Garantire equità tra i processi
✅ Eseguire i processi più urgenti prima

📌 Riassunto Finale in 3 Punti

1️⃣ Lo scheduling decide **quale processo usare la CPU e in che ordine**.
2️⃣ Esistono **tre tipi di scheduling**: lungo termine (gestisce la memoria), breve termine (sceglie il processo da eseguire), I/O (gestisce i dispositivi).

3️⃣ Gli **algoritmi di scheduling** determinano **l'ordine di esecuzione**, e possono basarsi su priorità, durata o turni fissi.

📊 Confronto tra le politiche di scheduling

Caratteristica	FCFS	Round Robin	SPN	SRT	HRRN
Metodo di scelta	Primo arrivato	Tempo fisso (quantum)	Processo più breve	Processo con tempo rimanente più breve	Processo con il punteggio più alto
Preemptive?	❌ No	✅ Sì	❌ No	✅ Sì	❌ No
Efficienza CPU	Bassa	Alta	Alta	Alta	Alta
Tempo di risposta	Alto se ci sono processi lunghi	Equo per tutti	Ottimizzato per processi brevi	Ottimizzato per processi brevi	Buon equilibrio
Overhead (tempo perso nei cambi di processo)	Minimo	Medio-Alto	Basso	Alto	Medio
Starvation possibile?	❌ No	❌ No	⚠️ Sì	⚠️ Sì	❌ No

📌 Scheduling nei Sistemi Multiutente

Nei sistemi con più utenti, bisogna garantire che tutti ricevano **una giusta quantità di CPU**.
✅ Si assegnano **priorità** ai processi, regolando il tempo CPU in base all'utilizzo.

📌 Scheduling in UNIX

- UNIX usa una formula per regolare la priorità dei processi.
- I processi con poco uso di CPU ricevono **più priorità**, per garantire equità.
- La priorità si aggiorna dinamicamente nel tempo.

✅ Questo sistema previene **starvation** ed è adatto ai sistemi con più utenti.

📌 Riepilogo finale

✔️ **FCFS** → Semplice, ma penalizza i processi brevi.
✔️ **Round Robin** → Ottimo per più utenti, ma può avere overhead elevato.
✔️ **SPN & SRT** → Ottimizzano il tempo medio, ma causano starvation.
✔️ **HRRN** → Equilibrio tra giustizia ed efficienza.
✔️ **UNIX** → Regola la priorità dinamicamente per evitare ingiustizie.

📌 INPUT/OUTPUT (I/O) NEI SISTEMI OPERATIVI

🔹 1. Cos'è l'I/O?

L'**Input/Output (I/O)** è il sistema che permette al computer di comunicare con dispositivi esterni. Ci sono tre tipi di dispositivi:

1. **Dispositivi leggibili dall'uomo** → (monitor, tastiera, mouse, stampante).
2. **Dispositivi leggibili dalla macchina** → (sensori, chiavette USB, dischi).
3. **Dispositivi di comunicazione** → (modem, schede di rete).

🔹 2. Caratteristiche dei dispositivi di I/O

Quando usiamo un dispositivo di I/O, dobbiamo considerare:

- 📏 **Velocità di trasferimento** → Ogni dispositivo ha una velocità diversa (es. un hard disk è più veloce di un modem).
- 📌 **Quanto viene usato** → Se il dispositivo è usato spesso o raramente.
- ⚙️ **Complessità di controllo** → Alcuni dispositivi hanno un proprio sistema di gestione (es. un disco rigido).
- 📦 **Modo di trasferimento**:
  - **Carattere per carattere** → (tastiera, mouse: inviano dati singolarmente).
  - **A blocchi** → (dischi e memorie esterne: inviano grandi quantità di dati insieme).

🔹 3. Il DMA (Direct Memory Access)

Il **DMA (Accesso Diretto alla Memoria (quella principale cioè la RAM))** permette di trasferire dati tra RAM e dispositivi di I/O **senza usare la CPU**. Questo è utile quando i dati da trasferire sono tanti.

⚙️ **Come funziona il DMA?**
1️⃣ Il **processore dice** al DMA di iniziare il **trasferimento**.
2️⃣ Il **DMA** prende il **controllo del bus di sistema** (senza coinvolgere la CPU) per gestire il **trasferimento** diretto **tra RAM e dispositivo di I/O**.

- **Perché?** Perché il DMA deve **spostare i dati** senza passare per la CPU, usando la RAM come punto di appoggio **se necessario**.
  3️⃣ Il **DMA trasferisce i dati** tra il dispositivo di I/O e la memoria RAM (o viceversa) attraverso il **bus di sistema**.
  4️⃣ Finito il trasferimento, il **DMA rilascia il bus di sistema e avvisa la CPU** con un segnale di interrupt.

🔹 Chiarimenti:

- Il **DMA non "prende il controllo della RAM"** nel senso di gestirla direttamente, **ma la utilizza come sorgente o destinazione dei dati**.
- Il **trasferimento** può **avvenire** in **entrambe** le **direzioni**:
  - Da un dispositivo di I/O → alla RAM
  - Dalla RAM → a un dispositivo di I/O
- Il **bus di sistema è essenziale** perché è il canale che permette il trasferimento dati senza passare dalla CPU.

- Il **bus di sistema** è un insieme di linee di comunicazione che collegano i componenti principali di un computer, come **CPU, memoria RAM e dispositivi di I/O**.
- Il **bus di sistema** comprende tre tipi di bus principali che LAVORANO INSIEME per permettere la COMUNICAZIONE: 1️⃣ **Bus dati** → Trasporta i dati tra CPU, memoria e periferiche.
  2️⃣ **Bus indirizzi** → Indica la posizione in memoria dove leggere o scrivere i dati.
  3️⃣ **Bus di controllo** → Gestisce i segnali di comando (lettura, scrittura, interrupt, ecc.).

🔹 4. Tipi di gestione dell'I/O

Ci sono tre modi per gestire l'I/O:

- 🖥️ **I/O programmato** → Il processore controlla tutto, ma è lento.
- 🔔 **I/O con interruzioni** → Il dispositivo avvisa la CPU quando ha bisogno di attenzione.
- ⚡ **I/O con DMA** → Il DMA trasferisce dati in autonomia, senza disturbare la CPU.

🔹 5. Obiettivi della gestione dell'I/O

🎯 **Efficienza** → Ridurre il lavoro della CPU.
🎯 **Modularità** → Separare la gestione dei dispositivi dal processore.
🎯 **Gestione unificata** → Un solo metodo per controllare tutti i dispositivi.

📚 **Conclusione:**
L'I/O è fondamentale per il funzionamento del computer. Il DMA aiuta a gestire i trasferimenti di dati senza sovraccaricare la CPU, migliorando l'efficienza del sistema.

📚 STRUTTURA E FUNZIONAMENTO DELL'I/O

1️⃣ Struttura dell'I/O

L'**Input/Output (I/O)** è il modo in cui il computer comunica con dispositivi esterni come **stampanti, tastiere e dischi rigidi**. Il sistema operativo gestisce queste operazioni in livelli, dal più astratto al più concreto.

📌 Livelli dell'I/O:

1️⃣ **Flusso di dati tra computer e dispositivi** 🖨️⌨️
• Es.: Stampanti, tastiere, dischi rigidi.
2️⃣ **Flusso di dati nei canali di comunicazione** 🌐🔌
• Es.: Reti, porte USB, Bluetooth.
3️⃣ **Flusso di dati nei file system** 📂💾
• Es.: Lettura/scrittura di file su disco.

🔧 Come il Sistema Operativo gestisce l'I/O:

✅ **Livello logico:** Comandi di base (**open, close, read, write**).
✅ **Traduzione per il dispositivo:** Trasforma i comandi logici in operazioni specifiche per ogni

dispositivo.
✅ **Schedulazione e controllo:** Organizza l'ordine delle operazioni per evitare rallentamenti.
✅ **Comunicazione:** Gestisce i protocolli di trasmissione dei dati.
✅ **Gestione fisica:** Si occupa della lettura e scrittura effettiva dei dati nei dispositivi.

2️⃣ Buffering (Memoria Temporanea) 🧠💾

Il **buffer** è un'area di memoria temporanea che **serve per velocizzare l'I/O**. Quando un dispositivo trasferisce dati, questi vengono prima salvati nel buffer e poi inviati alla destinazione finale.

📌 **Perché il buffering è utile?**
✅ Evita che il processore debba aspettare i dati.
✅ Migliora la velocità del sistema.
✅ Ottimizza la gestione dell'I/O nei dispositivi lenti (es. stampanti e dischi rigidi).

📌 **Tipologie di Buffer:**

🔹 **Buffer singolo** 🏗️

- Un solo buffer riceve i dati.
- Se il buffer è pieno, il sistema deve **aspettare** prima di continuare.

🔹 **Buffer doppio** 🏗️🏗️

- Un buffer viene usato per **leggere**, l'altro per **scrivere**.
- Permette di eseguire operazioni **contemporaneamente**, migliorando la velocità.

🔹 **Buffer circolare** 🔄🏗️🏗️🏗️

- Utilizza **più buffer** in sequenza.
- Quando un buffer è libero, viene **riutilizzato**, evitando tempi morti.

3️⃣ Prestazioni del Disco Rigido 💿⚡

Il **disco rigido** è uno dei dispositivi I/O più importanti. La sua velocità dipende da 3 **fattori chiave**:

📌 Questi 3 fattori che influenzano la velocità del disco sono:

1️⃣ **Tempo di ricerca (Seek Time)** ⏳

• Il **tempo** impiegato dalla **testina** (è il componente che legge e scrive i dati sulla superficie dei piatti (i dischi magnetici dentro gli hard disk). La testina si muove sopra il piatto per accedere alle informazioni memorizzate) **per spostarsi** dalla sua **posizione attuale** alla **traccia** (è una corsia circolare (una linea immaginaria) disegnata sulla superficie del piatto. I dati vengono organizzati lungo queste tracce, che sono disposte in maniera concentrica (una accanto all'altra) su ogni piatto) che contiene i **dati richiesti**.

2️⃣ **Tempo di latenza (Rotational Delay)** 🔄

• Il **tempo** necessario affinché il **settore** (la porzione del disco dove sono memorizzati i dati) **corretto giri** sotto la testina.

3️⃣ **Tempo di trasferimento (Data Transfer Time)** 🚀
• Il tempo impiegato per **spostare i dati** dal disco alla RAM.

📌 **Processo di lettura da un disco rigido:**

1️⃣ Il sistema invia il comando per accedere ai dati.
2️⃣ Il disco sposta la testina sulla traccia corretta (**seek time**).
3️⃣ Il disco gira fino a posizionare il settore giusto sotto la testina (**rotational delay**).
4️⃣ I dati vengono trasferiti alla RAM (**data transfer time**).

📌 **Trucchi per migliorare la velocità del disco**

✅ **Caching:** Salvare i dati più usati in memoria per un accesso più rapido.
✅ **Schedulazione degli accessi:** Organizzare la sequenza delle richieste per ridurre i tempi di ricerca.

✅ **CONCLUSIONI FINALI**

✔️ L'I/O è organizzato in **più livelli** per una gestione efficiente.
✔️ Il **buffering** evita rallentamenti e migliora la velocità del sistema.
✔️ Le prestazioni del **disco rigido** dipendono da **seek time, rotational delay e data transfer time**.
✔️ Ottimizzare l'uso del disco con **caching e schedulazione** migliora l'efficienza.

✅ **TEMPI IMPORTANTI DI UN DISCO RIGIDO**
📌 **1. Tempo di ricerca:** tempo che impiega la testina del disco per spostarsi sulla traccia giusta. (⏳ 10-20 ms)
📌 **2. Tempo di latenza rotazionale:** tempo che la testina aspetta finché il disco non ruota abbastanza per trovare il settore giusto. (⚙️ Circa 2 ms)
📌 **3. Tempo di trasferimento:** tempo necessario per trasferire i dati una volta che la testina è sulla traccia giusta.

✅ **Formula del tempo di trasferimento dei dati (Tr):**
📝 **Tr = b / r*N**
📌 Dove:
◦ **b** → bit da trasferire
◦ **r** → velocità di rotazione del disco (in giri/sec)
◦ **N** → numero di bit su ogni traccia

✅ **Formula del tempo di accesso totale (Ta):**
📝 **Ta = Ts + (1 / 2*r) + (b / r*N)**
📌 Dove:
◦ **Ts** → tempo medio di ricerca
◦ **1 / 2*r** → latenza rotazionale media
◦ **b / r*N** → tempo di trasferimento dati

✅ **Politiche di schedulazione del disco:**
🔄 Le politiche servono a decidere in che ordine servire le richieste di lettura/scrittura, per migliorare l'efficienza.

📌 **1. FIFO (First In, First Out)** 🔴 **Tempo lungo** quindi poco efficiente **ma semplice**
◦ Le **richieste** vengono gestite in **ordine di arrivo**.
◦ ⚠️ Può causare lunghe attese.

📌 **2. SSTF (Shortest Seek Time First)** 🟡 **Tempo breve, ma** rischio **starvation**
◦ La **testina serve** prima la **richiesta** più **vicina**.
◦ ⚠️ Le richieste lontane potrebbero aspettare a lungo.

📌 **3. SCAN (Algoritmo dell'ascensore)** 🟢 **Tempo bilanciato**
◦ La **testina** si **muove** in una **direzione fino** alla **fine** del **disco**, **poi torna** indietro.
◦ ✅ Riduce il tempo di attesa rispetto a FIFO e SSTF.

📌 **4. C-SCAN (Circular SCAN)** 🔵 **Tempo equo per tutte le richieste**
◦ La **testina** si **muove solo in una direzione** e **torna** all'**inizio senza servire richieste** nel ritorno.
◦ ✅ Tempo di attesa uguale per tutte le richieste.

📌 **5. N-Step-SCAN** 🟣 **Divide le richieste in sotto-code**
◦ **Evita** che alcune richieste restino in **attesa** troppo a **lungo**.
◦ ✅ Migliora la gestione del carico.

📌 **6. FSCAN** 🟠 **Coda stabile, evita ritardi**
◦ Le nuove **richieste** vengono messe in una **coda separata** e **servite** solo **dopo** che la **coda attuale** è **terminata**.
◦ ✅ Previene che nuove richieste ritardino quelle vecchie.

✅ 📊 **Grafico delle performance:**
📌 FIFO → 🚶 Lento con molte richieste
📌 SSTF → 🏃 Ottimo per richieste vicine
📌 SCAN & C-SCAN → ⚖️ Equilibrati per richieste distribuite
📌 N-Step-SCAN & FSCAN → 🔄 Migliorano la gestione delle richieste

---

GESTIONE DEI FILE NEI SISTEMI OPERATIVI

1. Cosa sono i file?

📁 Un **file** è un insieme di dati salvati nel computer.
Può contenere:

- **Testo** (documenti, codice).
- **Immagini** (foto, disegni).
- **Video e audio** (film, musica).

🔹 I file sono salvati su **dischi rigidi (HDD, SSD), DVD, chiavette USB** e altri dispositivi.
🔹 I file possono avere **permessi di accesso** per controllare chi può leggerli, modificarli o eliminarli.

---

2. File System

🗂️ Il **file system** è il software che organizza i file nel computer.
Funzioni principali:
✔️ **Memorizza e organizza i file** in cartelle e sottocartelle.
✔️ **Traduce i dati digitali** in una forma leggibile dal computer.
✔️ **Gestisce i permessi** per controllare chi può accedere ai file.
✔️ **Tiene traccia** di dimensioni, data di creazione e modifiche ai file.

📌 **Esempi di file system:**

- **NTFS, FAT32, exFAT** (Windows).
- **EXT4** (Linux).
- **HFS+, APFS** (Mac).

---

3. Tipi di Memorizzazione

🖴 **Hard Disk (HDD)**: Memoria magnetica con piatti rotanti che salvano dati sotto forma di 0 e 1.
⚡ **SSD**: Memoria a stato solido più veloce e resistente dell'HDD.
💿 **DVD**: Memoria ottica che salva dati sotto forma di incisioni sulla superficie del disco.

---

4. Sistema di Gestione dei File

💡 Permette agli utenti e ai programmi di accedere ai file in modo sicuro ed efficiente.
Operazioni principali:
✔️ Creare, leggere, modificare ed eliminare file.
✔️ Copiare e spostare file tra diverse cartelle o dispositivi.
✔️ Aprire e chiudere file.
✔️ Proteggere i dati e recuperare file danneggiati o cancellati.

🎯 **Obiettivi principali:**

- Garantire l'**accesso rapido ai file**.
- Evitare **errori e perdita di dati**.
- Ottimizzare la **velocità e l'uso dello spazio di memoria**.

---

5. Architettura del Sistema di Gestione dei File

🔹 **Driver dei dispositivi** → Comunicano con hard disk, SSD e altri dispositivi di memoria.
🔹 **Basic File System** → Organizza i blocchi di dati e gestisce la RAM.
🔹 **Logical File System** → Controlla i permessi di accesso e protegge i file.
🔹 **User Program** → Programmi che permettono all'utente di usare i file (Esplora File di Windows).

📌 Riassunto Finale

✅ Un **file** è un insieme di dati salvati nel computer.
✅ Il **file system** organizza i file e ne controlla l'accesso.
✅ I file possono essere salvati su **HDD, SSD, DVD e chiavette USB**.
✅ Il **sistema di gestione dei file** permette di creare, aprire, modificare e cancellare file.
✅ L'architettura del sistema di gestione dei file include **driver, gestione della memoria e permessi di accesso**.

📂 FILE SYSTEM E DIRECTORY

1️⃣ Cosa sono le **Directory**?

Le **directory** sono cartelle che organizzano i file nel computer.
Ogni file dentro una directory ha alcune informazioni importanti:

- **Nome del file** (unico nella directory).
- **Tipo di file** (testo, immagine, programma, ecc.).
- **Dimensione** (in byte o blocchi).
- **Dove si trova** sul disco (indirizzo, volume, settore).
- **Chi può accedere** (proprietario e permessi di lettura/scrittura).
- **Date importanti** (creazione, ultima modifica, ultimo accesso).

2️⃣ Come è organizzato un File System?

Il file system ha una **struttura ad albero** con due livelli principali:

- **Directory principale** (contiene altre directory e file).
- **Sotto-directory** (possono contenere altri file o altre cartelle).
- Ogni file ha un **indirizzo** e dei **diritti di accesso**.
- Ogni directory può contenere altre directory, creando una struttura a livelli.

3️⃣ Come vengono allocati i File?

Quando salviamo un file, il sistema lo divide in **blocchi** e lo assegna alla **memoria secondaria** (come hard disk, SSD, ecc.).
Esistono due metodi principali per questa operazione:

1. **Preallocazione** → Lo spazio viene riservato in anticipo (può sprecare memoria).
2. **Allocazione dinamica** → Lo spazio viene dato solo quando serve (più efficiente, ma può frammentare il disco).

Ci sono anche due problemi da considerare:

- **Blocchi troppo grandi** → Sprecano memoria.
- **Blocchi troppo piccoli** → Rallentano il sistema perché aumentano il numero di riferimenti.

4️⃣ Come si gestisce lo spazio libero?

Per sapere quali blocchi di memoria sono **liberi** o **occupati**, si usano tre metodi:

1. **Lista dei blocchi liberi** → Ogni file ha un puntatore all'ultimo blocco usato e il sistema segue la catena.
2. **Bitmap** → Una tabella con bit (1 = occupato, 0 = libero).
3. **File Allocation Table (FAT)** → Una tabella che indica in quale ordine sono collegati i blocchi di memoria di ogni file.

5️⃣ Come funziona UNIX?

- UNIX usa una **struttura gerarchica** con directory e sotto-directory.
- I file sono gestiti con **inode**, che contengono tutte le informazioni sul file.
- Per trovare un file, si usano **puntatori** che indicano dove sono i blocchi di dati.
- Ci sono due tipi di puntatori:
  - **Diretti** → Puntano subito ai blocchi di dati.
  - **Indiretti** → Puntano ad altri puntatori che a loro volta indicano i blocchi.

Chiarimento:

📌 Cosa è un inode?

Nel file system di UNIX, ogni file è gestito tramite un **inode** (abbreviazione di "index node").
Un **inode** (index node) è una **struttura dati** che contiene **tutte le informazioni su un file**, **esclusi**:

- Il nome del file.
- Il contenuto effettivo del file.

INFORMAZIONE AGGIUNTIVE (non chieste dal prof)

📄 Cosa contiene un inode?

Ecco i **metadati** memorizzati in un inode:

1. **Numero identificativo unico (cioè il numero inode)** → Ogni file ha un numero inode univoco. Il sistema usa il numero inode per riferirsi all'inode (che ha tutte le informazioni del file). Poi il sistema segue i puntatori nell'inode per leggere i dati effettivi del file.
2. **Tipo di file** → Indica se è un file normale, una directory, un collegamento simbolico, ecc.
3. **Permessi di accesso** → Specifica chi può leggere, scrivere o eseguire il file.
4. **Proprietario** (UID) e **gruppo** (GID) del file.
5. **Dimensioni** del file in byte.
6. **Timestamps (date importanti)**:
   - Data di creazione.
   - Ultima modifica.
   - Ultimo accesso.

7. **Numero di link fisici** (hard link) → Conta quanti nomi di file puntano allo stesso inode.
8. **Puntatori** ai blocchi del disco dove è memorizzato il contenuto del file.

🗂️ Come funziona il collegamento tra file e inode?

- Le **directory** sono file speciali che mappano **nomi di file** a **numeri di inode**.
- Quando apri un file, il sistema:
    1. Cerca il nome del file nella directory.
    2. Ottiene il numero di inode associato.
    3. Accede all'inode per leggere i metadati e trovare i blocchi di dati sul disco.

👉 **Esempio**:
Se hai un file `documento.txt`, la directory contiene una voce del tipo:
`12345 documento.txt`

🔗 Struttura dei puntatori in un inode

Per gestire file di grandi dimensioni, gli inode usano una **gerarchia di puntatori**:

- **Puntatori diretti**: Puntano direttamente a blocchi di dati (es: 12 puntatori diretti).
- **Puntatori indiretti**:
    - **Indiretto singolo**: Punta a un blocco di puntatori, che a loro volta puntano a dati.
    - **Doppio indiretto**: Due livelli di puntatori.
    - **Triplo indiretto**: Tre livelli di puntatori (per file enormi).

🔍 Come visualizzare gli inode?

Su terminale UNIX/Linux, usa il comando:

bash
```
ls -i nomefile.txt
```

Esempio di output:
`12345 nomefile.txt` → Il file `nomefile.txt` ha inode `12345`.

❗ Perché gli inode sono importanti?

1. **Efficienza**: Consentono un accesso rapido ai metadati senza leggere il contenuto del file.
2. **Gestione dell'hardware**: Ottimizzano l'uso del disco grazie ai puntatori diretti/indiretti.
3. **Sicurezza**: I permessi nell'inode controllano chi può modificare/leggere il file.

⚠️ Limitazioni degli inode

- Ogni file system ha un **numero massimo di inode** predefinito (es: creato con `mkfs`).
- Se tutti gli inode sono esauriti, non è possibile creare nuovi file, **anche se c'è spazio libero sul disco**.

Riassunto in punti chiave

✅ L'inode è la "carta d'identità" del file.
✅ Contiene metadati, non il nome o il contenuto.
✅ Le directory collegano nomi di file a numeri di inode.
✅ I puntatori diretti/indiretti gestiscono file di grandi dimensioni.
✅ Comando `ls -i` mostra il numero di inode.