SISTEMI OPERATIVI

📘 Capitolo 1 – Introduzione

Il sistema operativo è un programma che controlla l'esecuzione degli altri programmi e gestisce le risorse del computer. Ha due funzioni principali:

1. Fornire un'interfaccia tra applicazioni e hardware.
2. Gestire le risorse (CPU, memoria, dispositivi di I/O, file).

Gli obiettivi sono tre:

- Convenienza: rendere più semplice l'uso del computer.
- Efficienza: ottimizzare l'uso delle risorse.
- Evoluzione: permettere aggiornamenti per supportare nuovi dispositivi e servizi.

———

Compiti principali del sistema operativo

- Creazione, esecuzione e terminazione dei programmi.
- Gestione dei file e dei dispositivi di I/O.
- Controllo dell'accesso alle risorse e sicurezza.
- Rilevazione e gestione degli errori.
- Contabilità, cioè misurazione dell'uso delle risorse.

In sintesi: il sistema operativo agisce come gestore di risorse.

———

Evoluzione storica

- Prima generazione (1945–1955): computer a valvole, nessun sistema operativo; i programmi erano caricati manualmente da consolle.
- Seconda generazione (1955–1965): transistor, nascono i sistemi a lotti (batch systems). Un monitor residente gestisce job raggruppati. Linguaggio di controllo: JCL.
- Terza generazione (1965–1980): circuiti integrati, multiprogrammazione. Più programmi residenti in memoria, la CPU passa da uno all'altro durante le attese I/O → aumento dell'utilizzo della CPU.
- Quarta generazione (1980–oggi): personal computer, interfacce grafiche (GUI). Esempi: MS-DOS, Windows, UNIX, MacOS.
- Quinta generazione (1990–oggi): dispositivi mobili, smartphone, con risorse limitate e interfacce touch. Sistemi operativi come Android e iOS.

———

Multiprogrammazione e Time-Sharing

- Multiprogrammazione: obiettivo → massimizzare l'uso della CPU.

La CPU non rimane mai inattiva perché, se un processo è in attesa di I/O, un altro può usare il processore.

- Time-sharing: obiettivo → ridurre il tempo di risposta.

La CPU è divisa in "quanto" (time slice) e ogni utente collegato ha l'impressione di usare il sistema in esclusiva.

———

✅ In sintesi da ricordare per l'orale:

- SO = programma di controllo e gestore di risorse.
- Obiettivi = convenienza, efficienza, evoluzione.

- Evoluzione: nessun SO → batch → multiprogrammazione → PC con GUI → smartphone.
- Differenza chiave: multiprogrammazione (efficienza CPU) vs time-sharing (tempo di risposta per utenti interattivi).

📘 Capitolo 2 – Architettura del sistema di elaborazione

Il computer può essere visto come un insieme di componenti collegati da un bus di sistema:
- CPU (unità di elaborazione)
- Memoria principale (RAM)
- Moduli di I/O (periferiche)

All'interno della CPU troviamo registri fondamentali:
- PC (Program Counter) → contiene l'indirizzo della prossima istruzione da eseguire.
- IR (Instruction Register) → contiene l'istruzione in esecuzione.
- MAR (Memory Address Register) → contiene l'indirizzo della memoria da leggere/scrivere.
- MBR (Memory Buffer Register) → contiene il dato letto/scritto in memoria.
- Registri per I/O (I/O AR, I/O BR) → usati per indirizzi e dati delle periferiche.

Ciclo di istruzione (Fetch–Execute)

Ogni programma viene eseguito ripetendo ciclicamente due fasi:
1. Fetch: la CPU preleva dalla memoria l'istruzione puntata dal PC e la carica nell'IR. Poi incrementa il PC.
2. Execute: la CPU decodifica l'istruzione e la esegue (può essere elaborazione dati, I/O, salto di programma, ecc.).

Esempio:
- istruzione LOAD X → la CPU carica nel registro AC il contenuto della cella X.
- istruzione ADD Y → la CPU somma il contenuto della cella Y ad AC.
- istruzione STORE Z → il contenuto di AC viene scritto nella cella Z.

👉 Il ciclo fetch-execute è il cuore del funzionamento di un processore.

Interruzioni

Un concetto fondamentale per capire i sistemi operativi è l'interruzione (interrupt).
Permette a un modulo esterno (es. periferica I/O, timer) di segnalare alla CPU un evento, interrompendo temporaneamente il programma in esecuzione.

Tipi di interruzioni
- Di programma → errori (overflow, divisione per zero, accesso a indirizzi non validi).
- Timer → consente al SO di eseguire funzioni periodiche (es. cambiare processo).
- Di I/O → generate da un dispositivo al termine di un'operazione.
- Hardware → guasti fisici.

Come funziona
1. Il dispositivo segnala un'interruzione.
2. La CPU completa l'istruzione corrente.
3. Salva lo stato (registri, PC) in uno stack di controllo.
4. Carica il programma di gestione dell'interruzione (interrupt handler).
5. Alla fine, ripristina lo stato e continua l'esecuzione del programma interrotto.

Vantaggio: la CPU non rimane inattiva ad aspettare l'I/O.

———

Interruzioni multiple

Ci sono due modi per gestirle:

- Interruzioni disabilitate → una alla volta, semplice ma inefficiente.
- Schema a priorità → alcune interruzioni hanno precedenza (es. un errore hardware è più urgente di una tastiera).

Esempio pratico:

- se arriva un'interruzione da tastiera e subito dopo una dal disco, il sistema può decidere di servire prima il disco perché più critico.

———

Architettura dei sistemi operativi

Quando si progetta un sistema operativo bisogna considerare diversi aspetti:

- Processi → gestione e comunicazione.
- Memoria → allocazione e protezione.
- Sicurezza → protezione delle informazioni.
- Schedulazione e risorse → decidere quale processo eseguire e come assegnare CPU/memoria.
- Struttura del sistema → modello monolitico, stratificato, microkernel, ecc. (ripreso nei capitoli successivi).

———

Macchine virtuali

Un concetto chiave moderno: il sistema operativo può creare macchine virtuali.

- Ogni macchina virtuale sembra un computer a sé, con propria memoria e proprio sistema operativo.
- È gestita da un Virtual Machine Monitor (VMM), che coordina l'uso dell'hardware fisico.

Esempio:
Con VMware o VirtualBox, sul tuo PC puoi eseguire Linux e Windows contemporaneamente, ognuno crede di avere la CPU e la memoria tutta per sé.

———

✅ Cose da ricordare per l'orale:

- Componenti principali del sistema: CPU, memoria, I/O.
- Registri fondamentali (PC, IR, MAR, MBR).
- Ciclo fetch–execute: istruzione prelevata, decodificata, eseguita.
- Interruzioni: tipi (programma, timer, I/O, hardware), vantaggi, priorità.
- Architettura del SO: processi, memoria, sicurezza, risorse.
- Concetto di macchina virtuale.

📘 Capitolo 3 – Struttura del sistema operativo e servizi

Servizi del sistema operativo

Il sistema operativo si trova tra hardware e programmi utente e fornisce una serie di servizi fondamentali:

- Interfaccia utente
- Linea di comando (CLI)
- Interfaccia grafica (GUI)
- Modalità batch (insieme di comandi eseguiti in sequenza)
- Esecuzione dei programmi: caricamento, avvio, terminazione.
- Operazioni di I/O: astrae le periferiche (stampa, disco, tastiera…) con comandi uniformi.
- Gestione del file system: creazione, apertura, cancellazione, permessi.
- Comunicazione tra processi: memoria condivisa o scambio di messaggi.
- Rilevazione errori: controlli su hardware, I/O e programmi.
- Allocazione delle risorse: CPU, memoria, disco, dispositivi.
- Accounting: tiene traccia dell'uso delle risorse.
- Protezione e sicurezza: controllo degli accessi a file, memoria e dispositivi.

👉 Questi servizi sono accessibili tramite system call.

System call

Le chiamate di sistema sono l'interfaccia che i programmi usano per chiedere servizi al sistema operativo.

- Sono esposte tramite API (es. libreria C).
- I parametri possono essere passati in vari modi:
- nei registri,
- in un blocco di memoria (passando il puntatore),
- nello stack.

Esempio tipico (copia di un file):

1. Aprire il file di input → open()
2. Creare il file di output → creat()
3. Leggere dal file di input → read()
4. Scrivere nel file di output → write()
5. Chiudere i file → close()

Categorie di system call

1. Gestione dei processi: creare/terminare processi, caricare ed eseguire programmi, attendere eventi (fork, exec, wait).
2. Gestione dei file: creare, cancellare, aprire, chiudere, leggere e scrivere file.
3. Gestione dei dispositivi: richiedere e rilasciare dispositivi, leggere e scrivere dati, impostare attributi.
4. Gestione delle informazioni: leggere/settare data, ora, attributi di processi o file.
5. Comunicazione: invio/ricezione di messaggi, uso di memoria condivisa.
6. Protezione: gestione dei permessi e controllo degli accessi.

Programmi di sistema

Oltre al kernel, il SO include un insieme di utility che costituiscono l'ambiente di lavoro dell'utente:

- Gestione dei file (copie, editor di testo, comandi di ricerca)
- Supporto alla programmazione (compilatori, linker, debugger)
- Caricamento ed esecuzione programmi
- Comunicazioni in rete
- Servizi in background (demon o servizi di sistema)

Struttura del sistema operativo

Sistemi monolitici
- Tutte le funzioni sono contenute in un unico blocco.
- Vantaggio: velocità.
- Svantaggi: difficile da aggiornare e mantenere.
- Esempio: MS-DOS.

Sistemi stratificati
- Strutturati a livelli: ogni strato usa i servizi del livello inferiore e fornisce servizi al superiore.
- Vantaggi: semplicità di progettazione, portabilità, debugging.
- Svantaggi: overhead aggiuntivo.

Microkernel
- Il kernel contiene solo le funzioni di base (gestione processi, memoria, comunicazione).
- Altri servizi (file system, driver, ecc.) sono gestiti come processi in user mode che comunicano tramite messaggi.
- Vantaggi: affidabilità, portabilità, facilità di aggiornamento.
- Svantaggio: maggiore overhead.

Sistemi modulari
- Il kernel contiene solo le funzioni fondamentali.
- Altri servizi sono in moduli caricabili dinamicamente.
- Combina vantaggi di stratificazione e microkernel.
- Esempio: Solaris.

Avvio del sistema (bootstrap)
1. All'accensione, la CPU esegue un programma contenuto nella ROM: il bootstrap loader.
2. Questo carica in memoria il bootstrap program, che a sua volta carica il kernel.
3. Una volta avviato il kernel, il SO è operativo.

✅ Cose da ricordare per l'orale:
- Servizi fondamentali del SO (esecuzione programmi, I/O, file, comunicazione, protezione).
- Concetto e categorie di system call.
- Differenza tra sistemi monolitici, stratificati, microkernel e modulari.
- Funzione dei programmi di sistema.
- Fasi di avvio del sistema.

📘 Capitolo 4 – Processi

Definizione

Un processo è un programma in esecuzione.

- Programma → entità passiva (file eseguibile su disco).
- Processo → entità attiva, con program counter (PC), registri e risorse allocate.

Un programma può generare più processi (esempio: apri due volte la calcolatrice → due processi distinti).

Struttura del processo

Un processo in memoria è composto da:

- Sezione testo → il codice del programma.
- Sezione dati → variabili globali.
- Heap → per allocazione dinamica (malloc in C).
- Stack → variabili locali, parametri delle funzioni, indirizzi di ritorno.

Stati del processo

Un processo può trovarsi in diversi stati:

- New → appena creato.
- Ready → in attesa di essere eseguito dalla CPU.
- Running → in esecuzione.
- Waiting (o Blocked) → in attesa di un evento o I/O.
- Terminated → ha completato l'esecuzione.

Il passaggio tra stati è gestito dal sistema operativo (scheduler).

PCB (Process Control Block)

Ogni processo è rappresentato da una struttura dati chiamata PCB, che contiene:

- Stato del processo.
- PC e registri della CPU.
- Parametri di scheduling.
- Informazioni di memoria.
- Informazioni di accounting (tempo CPU, ID utente).
- Stato delle operazioni di I/O.

👉 Il PCB è salvato quando un processo viene sospeso e ricaricato quando riprende.

Code di scheduling

I PCB sono organizzati in diverse code:

- Job queue: tutti i processi del sistema.
- Ready queue: processi pronti per la CPU.
- Device queue: un processo per ogni dispositivo, contenente quelli in attesa di I/O.

Tipi di scheduler

- Scheduler a breve termine (CPU scheduler) → sceglie quale processo pronto assegnare alla CPU. Deve essere veloce (millisecondi).
- Scheduler a lungo termine (job scheduler) → decide quali job caricare in memoria. Controlla il grado di multiprogrammazione.
- Scheduler a medio termine → sospende e riattiva processi (swap in/out) per bilanciare carico e memoria.

Creazione dei processi

Un processo può creare un altro processo → gerarchia ad albero.

- Ogni processo ha un PID (process identifier).
- Il processo genitore può condividere risorse con il figlio oppure dargli risorse indipendenti.
- Il genitore può:
- eseguire in parallelo al figlio,
- oppure aspettare che il figlio termini (wait).

Terminazione dei processi

- Un processo termina con la system call exit().
- Restituisce un valore di stato al genitore (raccolto con wait()).
- Le risorse vengono deallocate.
- Il genitore può forzare la terminazione del figlio con abort().
- Attenzione: se il genitore termina prima del figlio → processo “orfano”.
- Se il figlio termina ma il genitore non fa wait() → processo zombie (rimane in tabella finché il padre non raccoglie lo stato).

✅ Cose da ricordare per l’orale:

- Differenza programma ↔ processo.
- Struttura del processo in memoria (testo, dati, heap, stack).
- Stati del processo e transizioni.
- PCB = descrittore del processo.
- Scheduler: breve, medio, lungo termine.
- Creazione processi: PID, gerarchia, fork/exec.
- Terminazione: exit, wait, zombie, orfani.

📘 Capitolo 5 – Threads

Processo vs Thread

- Processo = unità di allocazione delle risorse (memoria, file, I/O).
- Thread = unità di esecuzione (flusso di istruzioni assegnato alla CPU).

👉 Un processo può avere più thread che condividono risorse ma hanno stack e contesto separati.
Esempio: in un browser, un thread gestisce l'interfaccia, un altro il rendering delle pagine, un altro la rete.

———

Struttura

In un processo multithread:

- Spazio di indirizzamento e risorse → condivisi da tutti i thread.
- Ogni thread ha:
- Stato (pronto, attesa, esecuzione)
- Registri e contesto di CPU
- Stack (per funzioni e variabili locali)

———

Vantaggi dei thread

- Creazione e terminazione più rapida rispetto ai processi.
- Cambio di contesto tra thread dello stesso processo più veloce.
- Comunicazione semplice → condividono la memoria.
- Maggiore reattività (un thread può lavorare mentre altri aspettano I/O).
- Parallelismo: in un multicore, thread diversi possono eseguire in parallelo.

———

Stati e operazioni sui thread

- Stati: running, ready, waiting.
- Operazioni: creazione, sospensione, riattivazione, terminazione.

———

Sincronizzazione

Dato che i thread condividono memoria e risorse, serve sincronizzazione per evitare corruzione dei dati.
Esempio: due thread che aggiornano contemporaneamente un contatore senza mutua esclusione → risultato errato.
(Saranno approfonditi in concorrenza e mutua esclusione).

———

Implementazione dei thread

Thread a livello utente (user-level)

- Gestiti interamente nello spazio utente.
- Il kernel non sa che esistono.
- Vantaggi: rapidi, portabili, scheduling personalizzabile.
- Svantaggi: se un thread fa una system call bloccante, blocca tutto il processo.

Thread a livello kernel (kernel-level)

- Gestiti dal kernel, che conosce i thread.
- Vantaggi: un thread bloccato non ferma gli altri, possono girare su CPU diverse.
- Svantaggi: più overhead per cambi di modalità utente/kernel.

Approcci ibridi

- Thread utente mappati su un numero minore di thread kernel.
- Permettono il parallelismo senza bloccare l'intero processo.
- Esempio: Solaris.

———

Modelli di threading

- 1:1 → ogni thread utente = un thread kernel (UNIX tradizionale).
- M:1 → più thread utente mappati su un singolo thread kernel (vecchi sistemi).
- M:N → più thread utente mappati su più thread kernel (ibridi, più flessibili).

———

Multicore e legge di Amdahl

Il guadagno di velocità con più CPU è limitato dalla parte sequenziale del programma.

- Speedup massimo:

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{N}}$$

dove f = frazione parallela, N = numero processori.

Esempio: se il 90% del programma è parallelizzabile e usi 4 CPU, lo speedup massimo è ≈ 3.3, non 4.

———

Applicazioni che sfruttano i thread

- Applicazioni multithread (pochi processi, molti thread) → es. browser, IDE.
- Applicazioni multiprocesso (molti processi single-thread) → es. UNIX tradizionale.
- Java → la JVM è multithread e fornisce thread alle applicazioni.

———

Threads in UNIX/Linux

- Il kernel non distingue rigidamente processi e thread: entrambi sono "task".
- fork() crea un nuovo task con copia del PCB.
- clone() crea un task che può condividere strutture con il genitore (in base ai flag passati).

———

✅ Cose da ricordare per l'orale:

- Differenza processo ↔ thread.
- Vantaggi del multithreading.
- Thread utente vs thread kernel.
- Modelli 1:1, M:1, M:N.
- Legge di Amdahl → limite al parallelismo.
- Esempi: browser, Java, UNIX/Linux.

📘 Capitolo 6 – Concorrenza

Perché serve la concorrenza

In un sistema con multiprogrammazione o multithreading, più processi/thread possono eseguire in parallelo (su CPU diverse) oppure intercalati (su una sola CPU).
Questo porta vantaggi di utilizzo delle risorse, ma introduce un problema: la sincronizzazione.

Esempio:

- Due processi scrivono contemporaneamente sullo stesso file → risultato corrotto.
- Due thread aggiornano lo stesso contatore → valore errato.

Esecuzioni concorrenti

- Sequenziale: un processo alla volta. Nessun conflitto.
- Concorrente: più processi attivi nello stesso intervallo di tempo.
- Parallela: più processi in esecuzione simultanea (su multicore).

Problema della sezione critica

Ogni processo ha un segmento di codice che accede a una risorsa condivisa (variabile, file, memoria). Questo segmento è chiamato sezione critica.

Requisiti di una soluzione corretta

1. Mutua esclusione → un solo processo alla volta nella sezione critica.
2. Progresso → se nessuno è dentro, un processo deve poter entrare.
3. Attesa limitata (bounded waiting) → nessun processo deve aspettare all'infinito.

Soluzioni possibili

Approcci software

Algoritmi che garantiscono mutua esclusione senza supporto hardware, ma spesso inefficaci o inefficienti.

- Esempio: algoritmo di Dekker (sarà approfondito nel cap. 7).

Supporto hardware

- Disabilitare le interruzioni: impedisce il cambio di processo nella sezione critica → semplice ma inefficiente.
- Istruzioni atomiche (test-and-set, swap) → permettono di implementare lock.

Meccanismi di sincronizzazione

Semafori

Variabili speciali gestite dal SO:

- Semaforo binario (0 o 1) → come un lucchetto.
- Semaforo contatore → gestisce risorse multiple.

Operazioni:

- wait() → decrementa; se <0 il processo viene sospeso.
- signal() → incrementa; se ≤0 risveglia un processo sospeso.

Esempio classico: Produttore-Consumatore.

- Buffer condiviso.
- Produttore aggiunge dati solo se c'è spazio.
- Consumatore preleva dati solo se non è vuoto.
- Semafori garantiscono sincronizzazione.

Monitor

Costrutti dei linguaggi ad alto livello (es. Java).

- Racchiudono variabili e procedure di accesso.
- Consentono l'uso di variabili di condizione (wait, signal).
- Assicurano che un solo processo sia attivo nel monitor alla volta.

———

Problemi classici della concorrenza

- Produttore-Consumatore → gestione di un buffer condiviso.
- Lettori-Scrittori → lettori possono accedere contemporaneamente, ma scrittore deve avere accesso esclusivo.
- Filosofi a cena → gestione di risorse limitate (bacchette) per evitare deadlock.

———

Pericoli della concorrenza

1. Race condition → due processi accedono contemporaneamente a dati condivisi e il risultato dipende dall'ordine di esecuzione.
2. Deadlock (stallo) → più processi si bloccano in attesa di risorse (approfondito nel cap. 8).
3. Starvation → un processo non ottiene mai la risorsa perché altri hanno sempre la precedenza.

———

✅ Cose da ricordare per l'orale:

- Differenza sequenziale / concorrente / parallelo.
- Sezione critica + 3 requisiti (mutua esclusione, progresso, attesa limitata).
- Meccanismi: semafori, monitor.
- Problemi classici: produttore-consumatore, lettori-scrittori, filosofi a cena.
- Pericoli: race condition, deadlock, starvation.

📘 Capitolo 7 – Mutua esclusione

Algoritmi software

Algoritmo di Dekker

Primo protocollo per due processi senza supporto hardware:

- Ogni processo segnala l'intenzione di entrare nella sezione critica con una variabile flag.
- La variabile turn stabilisce a chi spetta il turno.
- Garantisce mutua esclusione, progresso e assenza di starvation.

Algoritmo di Peterson

Più semplice e usato come base teorica.

```
boolean flag[2];
int turn;

Processo P0:
while (true) {
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1); // attesa attiva
   // sezione critica
   flag[0] = false;
}
```

Funziona anche con più processori, ma è poco pratico per overhead e attesa attiva.

Semafori

Semaforo generico

- Variabile intera + coda di processi.
- Operazioni:
- wait(): decrementa; se <0 → il processo va in coda.
- signal(): incrementa; se ≤0 → risveglia un processo.

Semaforo binario

- Valore 0 o 1 (tipo lucchetto).
- Implementazione semplice della mutua esclusione.
- La coda dei processi in attesa deve evitare starvation → in genere FIFO.

👉 I semafori devono essere implementati con operazioni atomiche (test-and-set, swap).

Messaggi

Altro metodo di sincronizzazione:

- Primitive send(dest, msg) e receive(src, msg).
- Utile nei sistemi distribuiti, ma anche con memoria condivisa.

Tipi di comunicazione:

- Bloccante (mittente o ricevente restano in attesa).
- Non bloccante.
- Indirizzamento diretto (esplicito o implicito) o indiretto (tramite mailbox).

Problemi classici

Produttore-Consumatore

- Buffer condiviso.
- Condizioni: mutua esclusione + non scrivere se pieno + non leggere se vuoto.

Esempio con semafori generici:

```
semaphore n = 0, s = 1, e = buffer_size;

producer() {
  while (true) {
    produce();
    wait(e);  // spazio libero
    wait(s);  // lock
    append();
    signal(s);
    signal(n); // nuovo elemento
  }
}

consumer() {
  while (true) {
    wait(n);  // c'è un elemento
    wait(s);  // lock
    take();
    signal(s);
    signal(e); // spazio libero
    consume();
  }
}
```

Lettori e Scrittori

- Lettori: possono leggere in parallelo.
- Scrittore: deve avere accesso esclusivo.
- Varianti:
- Priorità ai lettori.
- Priorità agli scrittori.

Monitor

Costrutto nei linguaggi ad alto livello (es. Java).

- Variabili locali + procedure + variabili di condizione (wait, signal).
- Garantisce mutua esclusione automatica.
- Esempio: monitor per buffer limitato con notfull e notempty.

Filosofi a cena (cenno)

- 5 filosofi attorno a un tavolo, ciascuno con una bacchetta a destra e una a sinistra.
- Devono prendere due bacchette per mangiare → rischio di deadlock se tutti prendono la bacchetta sinistra insieme.
- Soluzioni: ordinamento, semafori, monitor.

Riassunto da fissare

- Algoritmi di Dekker e Peterson → esempi teorici di mutua esclusione software.
- Semafori → wait/signal, binari e contatori.
- Messaggi → send/receive, indirizzamento diretto o tramite mailbox.
- Problemi classici → produttore-consumatore, lettori-scrittori, filosofi a cena.
- Monitor → soluzione ad alto livello con variabili di condizione.

📘 Capitolo 8 – Stallo (Deadlock)

Definizione

Uno stallo (deadlock) si verifica quando un insieme di processi è bloccato perché ognuno aspetta una risorsa detenuta da un altro processo del gruppo.

- Nessuno può progredire.
- Nel caso generale, non esiste una soluzione efficiente.

Esempio semplice:

- Processo P vuole risorsa A, poi B.
- Processo Q vuole risorsa B, poi A.

→ entrambi rimangono bloccati.

———

Categorie di risorse

- Risorse riutilizzabili → CPU, memoria, file, semafori (possono essere rilasciate e riutilizzate).
- Risorse consumabili → messaggi, segnali, interruzioni (una volta usate spariscono).

———

Condizioni per lo stallo

Perché si verifichi deadlock, devono valere tutte e 4 le condizioni:

1. Mutua esclusione: una risorsa può essere usata da un solo processo.
2. Possesso e attesa: un processo che possiede una risorsa può chiederne altre.
3. Assenza di prerilascio: le risorse non possono essere tolte forzatamente a un processo.
4. Attesa circolare: esiste una catena di processi in cui ciascuno aspetta una risorsa detenuta dal successivo.

👉 Le prime 3 rendono il deadlock possibile; con la quarta diventa inevitabile.

———

Rappresentazione con grafi

Si usa il grafo di allocazione delle risorse:

- Nodo processo (P).
- Nodo risorsa (R).
- Freccia da processo a risorsa = richiesta.
- Freccia da risorsa a processo = risorsa assegnata.
- La presenza di un ciclo nel grafo = possibile deadlock.

Esempio: l'incrocio stradale dove ogni macchina blocca l'altra.

———

Strategie contro lo stallo

1. Prevenzione

Si progettano regole che impediscono a priori il verificarsi delle 4 condizioni.

- Mutua esclusione: non eliminabile per risorse fisiche.
- Possesso e attesa: un processo deve richiedere tutte le risorse insieme (inefficiente).
- Assenza di prerilascio: si può forzare il rilascio di risorse quando si aspetta altre (fattibile solo per alcune).
- Attesa circolare: imporre un ordine alle richieste (ogni processo può chiedere risorse solo in ordine crescente).

2. Evitamento

Si accettano le richieste solo se mantengono il sistema in uno stato sicuro.

- Algoritmo del banchiere (Banker's algorithm): simula l'assegnazione e controlla se esiste una sequenza di esecuzione che porta tutti i processi a completare.
- Se la richiesta porta a uno stato insicuro, viene rifiutata.

3. Rilevazione e ripristino

- Si permette che lo stallo si verifichi, ma il SO lo rileva periodicamente (controllo dei cicli nel grafo o uso di matrici).
- Una volta rilevato:
- Terminare i processi coinvolti.
- Oppure revocare risorse finché lo stallo si risolve.
- Possibile applicare politiche (terminare quello con priorità più bassa, con meno risorse allocate, ecc.).

———

Vantaggi e limiti

- Prevenzione → semplice, ma molto restrittiva.
- Evitamento (banchiere) → più flessibile, ma richiede conoscere in anticipo le richieste massime di ogni processo.
- Rilevazione → consente massima libertà, ma richiede meccanismi di recupero costosi.

———

Riassunto da fissare

- Deadlock = processi bloccati in attesa circolare di risorse.
- Condizioni necessarie: mutua esclusione, possesso e attesa, assenza di prerilascio, attesa circolare.
- Strategie:
- Prevenzione (eliminare almeno una condizione).
- Evitamento (algoritmo del banchiere).
- Rilevazione e ripristino.