Architetture dei sistemi operativi:
Un sistema grande e complesso quale un sistema operativo può essere realizzato in molti modi diversi, Tradizionalmente, il sistema era sviluppato come un unico blocco (sistema **monolitico**) che esegue tutte le funzioni del sistema dalla gestione della CPU all'interfaccia utente. Non vi è alcun ordinamento tra le funzioni del sistema: tutte le funzioni e le strutture dati sono accessibili da qualsiasi punto del kernel. Manutenzione ed espansione sono molto difficili: ad esempio un bug in una funzione potrebbe causare il blocco dell'intero sistema, e trovarlo e risolverlo potrebbe essere difficile poiché il codice è molto integrato. Il vantaggio principale di quest'approccio è che il sistema è compatto, veloce ed efficiente, quindi è un approccio adatto a sistemi semplici.
Un'evoluzione è il sistema a **struttura gerarchica,** che organizza il sistema su livelli funzionali. Una funzione di un certo livello può chiamare solo funzioni di livello inferiore. Non vi è comunque una separazione tra le componenti del SO. Distinguere le dipendenze gerarchiche potrebbe non essere immediato. Manutenzione ed espansione rimangono difficili, ma sono più gestibili di un sistema monolitico.
Un primo approccio modulare è il sistema **stratificato**.
Il sistema è scomposto in un certo numero di livelli (il più basso è l'hw). Ogni livello è un modulo che implementa un componente del SO (gestione CPU, memoria, …), nasconde i propri dettagli implementativi agli altri e può comunicare con il livello sottostante attraverso un'interfaccia ben definita.
Lo sviluppo e il debug del sistema sono facilitati: quando si implementa un nuovo livello, si presume che quelli sottostanti funzionino correttamente, poiché sono già stati testati; e dopo aver implementato e testato anche il nuovo livello, si può passare a un livello superiore in modo analogo.
I vantaggi quindi sono la modularità e la facilità di sviluppo e manutenzione.
Gli svantaggi sono la possibile difficoltà nell'identificare i livelli e l'efficienza limitata: una chiamata a una funzione di un certo livello deve passare attraverso tutti i livelli sopra ad esso, riducendo le prestazioni del sistema.
Ulteriore evoluzione è l'approccio a **microkernel,** che struttura il sistema rimuovendo dal kernel tutte le funzioni non indispensabili, implementandole come servizi a livello utente. Il kernel così creato è molto piccolo e ha solo funzioni minime di gestione processi, memoria e comunicazione. Lo scopo principale del microkernel è di permettere la comunicazione tra programmi client e servizi, oltre che tra i servizi stessi così che possano richiedersi funzioni l'un l'altro. La comunicazione avviene tramite messaggi scambiati dal kernel, quindi c'è un grande sovraccarico di gestione, che incide sull'efficienza del sistema. Questo modello permette la massima separazione tra meccanismi e politiche (è sempre bene separare il "come si fa" dal "cosa si fa"), una facile modificabilità, la massima portabilità (basta reimplementare le poche funzioni del microkernel utilizzate dai servizi per far funzionare il sistema su una macchina diversa) e affidabilità: se un servizio fallisce, il resto del sistema rimane inalterato, e si può riavviare il servizio.
Struttura modulare:
Quest'approccio coinvolge l'uso di tecniche di programmazione a oggetti, ed è il più utilizzato.
Come nell'approccio a microkernel, il kernel possiede solo un insieme minimo di funzioni, ma quì si collega dinamicamente ai moduli che implementano le varie funzioni del sistema al boot e durante l'esecuzione. Come nel modello a strati, le implementazioni di ogni modulo sono nascoste agli altri ma hanno interfacce ben definite per comunicare. La comunicazione è diretta, migliorando così le prestazioni rispetto all'approccio a microkernel (rimangono però limitate). Gli altri vantaggi di quest'implementazione sono gli stessi dei sistemi a microkernel.
Sistema a macchine virtuali (VM):
Il kernel di un sistema a VM si occupa di fornire una copia esatta dell'hardware sottostante ad ogni macchina virtuale eseguita in modalità utente, ognuna delle quali esegue un SO (anche diversi tra loro). In questo modo, si può fornire l'illusione a ogni utente di avere una macchina propria: schedulazione e memoria virtuale si occupano di suddividere tempo di cpu e memoria tra le VM, mentre spooling e file system gli forniscono periferiche e dischi virtuali. Un ulteriore vantaggio è l'isolamento di ogni macchina virtuale dalle altre e dalla macchina reale, che le rende adatte a testare delle modifiche a un

sistema operativo senza causare tempi di inattività nel sistema. Sfortunatamente questo impedisce anche la condivisone diretta di dati e risorse tra le VM. È però possibile condividere un disco virtuale tra le VM, o creare una rete virtuale tra esse che gli consenta di comunicare pur rimanendo completamente isolate.
Lo svantaggio principale di questi sistemi è il calo di prestazioni dovuto allo strato di virtualizzazione: se ad esempio un processo in modalità utente virtuale esegue una richiesta di IO, questa dovrà passare prima al kernel del SO di quella VM, quindi al software di controllo della VM (che dovrà eventualmente interpretare la richiesta), e infine il kernel del sistema VM, in modalità reale fisica, potrà eseguire la richiesta. Questo riduce molto le prestazioni.

Tradizionalmente, i processi fanno uso di un singolo flusso di controllo che svolge tutte le attività dell'applicazione; tuttavia ci sono situazioni in cui si vogliono eseguire attività simili contemporaneamente, o in cui si vuole continuare ricevere richieste nonostante altre stiano venendo servite. Tradizionalmente, si utilizza la chiamata di sistema fork per creare nuovi processi e le tecniche di IPC per condividere i dati: questo comporta un enorme spreco di memoria e tempo di CPU poiché la fork fa una copia dello spazio di indirizzamento del padre, anche se tutti i figli creati avranno bisogno di eseguire lo stesso codice, e di accedere agli stessi dati condivisi.
La soluzione è di introdurre i thread. Un thread è l'unità base di utilizzo della CPU. Può essere visto come un flusso di controllo all'interno di un processo, e ve ne possono essere molti per ogni processo. Possiede un id, dei suoi registri, un program counter, uno stack, se necessario dei dati privati, e condivide spazio di indirizzamento e risorse con gli altri thread del processo.
L'utilizzo dei thread porta a diversi benefici:

- Maggior disponibilità: è infatti possibile gestire delle richieste mentre altri thread sono occupati
- Condivisione delle risorse: tutti i thread condividono tra loro codice, dati e risorse, e ciò permette loro di accedervi semplicemente e anche di comunicare (sincronizzando però gli accessi per evitare problemi di consistenza)
- Economia: creare un nuovo thread richiede meno tempo e memoria rispetto a un nuovo processo poiché non è necessario copiare nulla
- Sfruttamento delle architetture multiprocessore: un kernel che supporta i thread può schedulare più thread dello stesso processo su processori diversi contemporaneamente, migliorando le prestazioni

Il supporto ai thread può essere fornito nello spazio utente (thread livello utente) o nel kernel (thread livello kernel).
Una libreria di thread fornisce al programmatore le funzioni per la creazione e la gestione dei thread. Per i thread livello utente, tutto il codice e le strutture dati della libreria si trovano nello spazio utente e le chiamate alle sue funzioni sono semplici chiamate a procedura. Per i thread livello kernel, invece, codice e dati si trovano nella memoria del SO, e le chiamate si traducono in chiamate di sistema.
Modelli multithread:
I thread utente di un'applicazione possono essere mappati sui thread kernel in diversi modi.
Modello molti a 1:
Tutti i thread utente sono mappati su un solo thread kernel. La libreria livello utente ha quindi il compito di simulare un ambiente multithread schedulando i thread utente sul thread kernel. Questo modello è semplice da realizzare ma presenta 2 problemi: poiché vi è un solo thread kernel, non sfrutta le architetture multiprocessore (il SO vede solo un processo da schedulare); inoltre la concorrenza è minima: se ad esempio un thread utente esegue una chiamata bloccate, bloccherà anche l'unico thread kernel e quindi l'intero processo.
Modello 1 a 1:
È il modello più utilizzato, e mappa ogni thread livello utente su un thread a livello kernel. Permette il massimo sfruttamento dei sistemi multiprocessore e la massima concorrenza (se un thread si blocca, si blocca solo quello); tuttavia l'overhead per la creazione un thread kernel per ogni thread utente va ad intaccare le prestazioni del sistema all'aumentare dei thread.
Modello molti a molti:
n thread utente sono mappati su m<=n thread kernel. In questo modo si ha sia lo sfruttamento delle architetture multiprocessore sia un'alta concorrenza (se un thread kernel si blocca, è sufficiente schedularne un altro per l'esecuzione), e permette agli sviluppatori di usare il numero di thread utente che desiderano. La libreria livello utente si occuperà di schedularli sui thread kernel disponibili.
Una variante è il modello a 2 livelli, che permette di mappare un thread utente su un thread kernel dedicato. È una modalità utile per gestire thread coordinatori.
Cooperazione tra thread:
La cooperazione può essere organizzata secondo vari modelli:

- Thread simmetrici: tutti i thread possono svolgere lo stesso insieme di attività. Quando arriva una richiesta, un qualsiasi thread disponibile può gestirla.

- Thread gerarchici:un thread coordinatore riceve le richieste, e da ordini a più thread lavoratori che le eseguono
- Thread in pipeline: come in una catena di montaggio, ogni thread è specializzato nell'eseguire una piccola operazione velocemente, passando poi il risultato parziale al thread successivo, fino ad ottenere il risultato completo. Permette un elevato throughput.

Problemi di gestione dei thread:
Fork ed exec all'interno di un thread:
In un processo tradizionale, la chiamata fork crea una copia del processo, ma in un'applicazione multithread si può comportare in 2 modi: può duplicare tutti i thread del processo chiamante, oppure solo il thread chiamante. Se è possibile scegliere, si sceglie in base alle istruzioni successive: se subito dopo c'è una exec, il nuovo processo sarà sovrascritto quindi basterà duplicare solo il thread chiamante; in caso contrario può essere meglio duplicarli tutti.
Cancellazione di un thread:
Cancellare un thread significa terminarlo prima che abbia completato la sua esecuzione. Può avvenire in 2 modi:

- Cancellazione asincrona: il thread viene terminato immediatamente, ma potrebbe lasciare risorse in stato inconsistente
- Cancellazione differita: si imposta una variabile nel thread target che questi deve controllare periodicamente in punti sicuri (punti di cancellazione), in cui non si rischi di lasciare risorse in stato inconsistente. Permette però a un thread di ignorare la richiesta

Gestione dei segnali e dei messaggi:
Quando arriva un segnale ad un processo multithread, si crea un problema: a quale thread va consegnato? Si può:

- Inviarlo al thread cui si applica (per segnali sincroni, come la divisione per 0)
- Inviarlo a tutti i thread del processo destinatario
- Inviarlo al sottoinsieme di thread che non bloccano quel segnale
- Designare un thread specifico che riceva e gestisca tutti i segnali

La soluzione da adottare dipende dall'applicazione. Tipicamente il SO consegna al primo thread che non lo blocca, così che sia gestito una sola volta.
Processi leggeri:
Nei sistemi che implementano i modelli molti a 1, molti a molti e a 2 livelli si introduce una struttura dati intermedia tra un thread livello kernel e i thread livello utente che devono essere mappati su esso: il lightweight process (LWP), che separa le 2 visioni e memorizza dei dati di gestione. Dal punto di vista del kernel, un LWP è visto come un processo, e come tale viene schedulato, con la particolarità che tutti gli LWP appartenenti a un processo hanno lo spazio di indirizzamento condiviso. Dal punto di vista della libreria utente, un LWP appare come un processore virtuale su cui schedulare l'esecuzione dei thread utente in base a qualche algoritmo. La libreria, inoltre, può variare il numero di thread a livello kernel (e quindi LWP) così da avere sempre un certo numero di thread in esecuzione.

Nei sistemi multitasking, la schedulazione ha il compito di gestire la turnazione dei processi sul processore in base a delle politiche. Lo scheduler della CPU ordina la lista dei processi pronti, ponendo in prima posizione quello che il dispatcher manderà in esecuzione. È importante che sia efficiente, ma soprattutto veloce poichè viene chiamato molto di frequente e si vuole minimizzare il sovraccarico di gestione. Esistono altri 2 livelli di schedulazione (a lungo e a medio termine), qui non trattati.
L'attivazione dello schedulatore può avvenire:

- Senza preemption: la cpu viene assegnata ad un processi finchè questi non la rilascia esplicitamente o si mette in attesa di un evento/periferica; quindi il controllo torna allo schedulatore. In questo caso l'attivazione è sincrona con l'evoluzione della computazione del processo
- Con preemption: il sistema può anche forzare il rilascio della cpu assegnata a un processo. Se ciò accade, l'attivazione è asincrona con la computazione del processo e se questi condivide risorse con altri, deve usare tecniche di sincronizzazione per evitare inconsistenze.

La scelta di un algoritmo di schedulazione va fatta in base ai criteri che si vogliono ottimizzare. Questi criteri sono:

- Sfruttamento del processore
- Throughput: ossia quanti processi vengono completati in un'unità di tempo.
- Tempo di turnaround: ossia il tempo che passa dal momento in cui un processo entra nel sistema e il momento in cui termina la sua esecuzione. Conta anche il tempo speso nelle varie code di attesa.
- Tempo di attesa: il tempo che un processo passa nella coda dei processi pronti.
- Tempo di risposta: il tempo che passa dal momento in cui viene formulata una richiesta e il momento in cui viene presentato il primo risultato.

Si desidera massimizzare i primi 2 e minimizzare gli ultimi 3. Inoltre, si desidera minimizzarne la varianza, così da rendere il sistema più predicibile.
Per valutare la bontà di un algoritmo di schedulazione nel contesto di applicazione si può procedere in vari modi.
Valutazione analitica:
Un tipo di valutazione analitica è la modellazione deterministica, che definisce in modo matematico le prestazioni di un algoritmo in base a un carico di lavoro prestabilito.
E' un metodo semplice, veloce e preciso, ma i risultati non sono generalizzabili, e nella realtà il carico di lavoro non è costante.
Valutazione statistica:
La valutazione statistica permette di avere dei risultati con un grado di incertezza associato.
Un esempio è la modellazione a reti di code: fa uso di distribuzioni di picchi di CPU e IO, e vede il computer come una rete di servizi, ognuno con una coda associata. Studia quindi i parametri di queste code (lunghezza media, tempi di attesa, …). Questo metodo è piuttosto veloce, ma richiede alcune semplificazioni che potrebbero discostarsi troppo dalla realtà.
Un altro tipo di valutazione statistica è la simulazione: si realizza un modello software del sistema, il più completo possibile, e lo si utilizza per testare il comportamento dell'algoritmo. I dati in input alla simulazione sono picchi di CPU e IO generati casualmente o la traccia di un sistema reale. Sono piuttosto accurate, ma onerose da realizzare.
Implementazione:
Se si vuole maggiore accuratezza, si può procedere con l'implementazione nel sistema reale, utilizzando gli strumenti di rilevazione del sistema per valutare l'efficienza dell'algoritmo.
E' il metodo più accurato, ma richiede tempo per la realizzazione e la cooperazione da parte degli utenti. Inoltre, per ottenere le prestazioni migliori nei vari momenti può essere necessario implementare più algoritmi di scheulazione, e si deve decidere se analizzare i casi singolarmente o studiare i valori medi.
Politiche di schedulazione:
FCFS:
Gestisce la coda dei processi pronti con una politica FIFO: i processi ottengono la CPU nell'ordine con cui

entrano nella coda. E' di tipo non preemptive. È una politica facile da implementare e molto veloce da eseguire, tuttavia il tempo di attesa medio dipende fortemente dall'ordine di arrivo dei processi (effetto convoglio).
Priorità:
Permette di associare ad ogni processo un indice di priorità definito internamente (in base a parametri misurabili), oppure esternamente (dall'utente o dall'amministratore del sistema), quindi ordina la coda in base a questi indici. Può essere realizzata con o senza preemption: nel primo caso quando un processo diventa pronto, richiede schedulazione, e se ha priorità maggiore rispetto a quello in esecuzione, lo sostituisce; nel secondo caso invece la schedulazione viene eseguita quando la cpu viene rilasciata.
Un problema di queste politiche è la possibilità di starvation, ossia il blocco indefinito di processi a priorità bassa poiché ci sono sempre processi a priorità maggiore ad ottenere la CPU. Si può risolvere introducendo l'aging, ossia aumentando la priorità di un processo mentre attende. In questo modo, anche i processi a priorità bassa saranno sicuramente eseguiti, dopodiché la priorità tornerà al valore iniziale.
Un caso particolare è la schedulazione SJF (shortest job first), in cui la priorità è l'inverso della durata del successivo picco di cpu di un processo. Minimizza il tempo di attesa medio, ma è necessario stimare la durata del picco in base alla storia passata poiché non è nota, e la stima potrebbe non essere precisa.
Round robin:
È la politica tipica dei sistemi time sharing e permette di distribuire uniformemente il tempo della cpu tra n processi in modo che ognuno ne ottenga 1/n. E' di tipo preemptive.
Si implementa impostando un timer per generare un interrupt allo scadere di un quanto di tempo prestabilito ogni volta che si assegna la cpu a un processo. Se questi non la rilascia la entro la fine del quanto, subisce preemption, torna in coda e lo schedulatore sceglie un nuovo processo da eseguire. Se la turnazione è abbastanza veloce, permette di creare l'illusione di esecuzione parallela. La velocità di esecuzione dei processi dipende dal numero di processi in coda; il tempo di turnaround dipende dalla durata del quanto di tempo. Il problema è deciderne la durata: più lungo riduce il sovraccarico di gestione ma anche l'illusione di parallelismo fino a degenerare in FCFS; più breve aumenta l'illusione di parallelismo, ma anche il sovraccarico di gestione. Idealmente, si vuole che circa l'80% dei picchi di CPU si esaurisca entro un quanto di tempo.
Coda a più livelli (C+L):
Se è possibile raggruppare i processi per tipologie omogenee, si può spezzare la coda dei processi pronti in più code, una per tipologia, così da poter scegliere l'algoritmo di schedulazione più adatto per ogni tipo di processo. Le code sono poi schedulate con un algoritmo dedicato (ad esempio condividendo il tempo tra le code).
I processi sono assegnati permanentemente ad una coda, tuttavia il loro "comportamento" può variare nel tempo, e quindi anche la coda più adatta. La variante con retroazione (C+LR), permette ai processi di migrare da una coda a un'altra, definendo delle politiche di promozione, di degradazione e di allocazione dei processi. In questo modo si può far si che ogni processo si trovi sempre nella coda più adatta al suo attuale compito.
È l'algoritmo più efficiente ma anche il più complesso.

Quando più processi accedono contemporaneamente alla stessa risorsa fisica o logica, si ha la concorrenza. Se l'esito delle operazioni dipende dall'ordine in cui sono eseguite, si ha una corsa critica, e ciò può causare inconsistenza nella risorsa. Una porzione di codice che causa corse critiche se eseguita in modo concorrente è detta sezione critica. Ciò va evitato con meccanismi e politiche di sincronizzazione.
Una soluzione al problema delle sezioni critiche deve soddisfare 3 condizioni:

- Mutua esclusione: un solo processo per volta deve eseguire la sua sezione critica
- Progresso: la competizione per entrare nella sezione critica deve avvenire tra i processi che non sono nella loro sezione critica, e la competizione non deve durare indefinitamente
- Attesa limitata: un processo non deve essere costretto ad attendere indefinitamente per entrare nella sua sezione critica

Vediamo ora alcune soluzioni.
Variabili di turno:
Si tratta di un approccio a livello di istruzioni. Una variabile di turno è una variabile condivisa tra i processi che stabilisce quale può usare una risorsa.
Si possono usare per risolvere il problema delle sezioni critiche. Esempio per 2 processi/thread:
si usano 2 flag inizializzati a false che rappresentano quando un processo è o vuolr entrare nella propria sezione critica; e una variabile di turno, inizializzata ad uno dei 2 processi. Quando un processo vuole entrare nella propria sezione critica, imposta il turno sull'altro e il proprio flag a true, e attende se e solo se il turno è quello dell'altro, e questi ha il flag a true; al rilascio imposterà il proprio flag a false. Grazie alla doppia condizione, se entrambi tentano di entrare contemporaneamente, solo uno dei 2 ci riuscirà, a seconda del valore finale assunto della variabile di turno, che decide chi entrerà per primo. In questo modo si rispettano le 3 condizioni e si risolve il problema della sezione critica.
Questo metodo all'aumentare dei processi scala male: serve infatti modificare il codice o scrivere codice che si adatti.
Variabili di lock:
E' un altro approccio a livello di istruzioni, ma non sono i processi ad alternarsi, bensì la risorsa ad avere una variabile che dica se è disponibile (0) oppure in uso (1). Questo metodo è indipendente dal numero di processi. Quando un processo vuole utilizzarle la risorsa, deve ottenere un lock in questo modo:

- Disabilitare gli interrupt
- Leggere la variabile di lock
- Se è a 0, la imposta ad 1, riabilita gli interrupt e usa la risorsa, altrimenti riabilita gli interrupt e si mette in attesa

La disabilitazione/riabilitazione degli interrupt rende la sequenza atomica, evitando una corsa critica se più processi tentassero di ottenere il lock contemporaneamente. Molti processori forniscono un supporto hardware, l'istruzione TEST-AND-SET, che esegue proprio questa sequenza di istruzioni in un'istruzione hardware, che per definizione è indivisibile.
Al rilascio, il lock viene reimpostato a 0.
Semafori binari e generalizzati:
Un semaforo è una struttura dati gestita dal SO che nel caso binario può assumere 2 valori: 1 (risorsa libera), 0 (risorsa occupata). Il sistema fornisce 2 istruzioni per utilizzare un semaforo:

- Acquire(S), usata per richiedere una risorsa: controlla il valore e se è 0 mette in attesa il processo, altrimenti lo imposta a 0 e il processo può usare la risorsa
- Release(S), usata per rilasciare la risorsa: reimposta il valore a 1

Il funzionamento è quindi simile a quello delle variabili di lock, ma sono gestiti dal SO.
I semafori generalizzati funzionano in modo analogo, e permettono di gestire istanze multiple delle risorse: possono assumere qualsiasi valore >=0, inizializzato al numero di istanze disponibili.
Le operazioni sono:

- Acquire(S), controlla il valore e se è 0 mette in attesa il processo, altrimenti sottrae 1 al valore e il processo può usare la risorsa
- Release(S), aumenta di 1 il valore

Un possibile problema riguarda il modo con cui un processo si mette in attesa: un ciclo infinito che

controlla il valore del semaforo (busy wait). Sebbene questo sia semplice da realizzare, in caso di attese lunghe si spreca il tempo del processore inutilmente. Si può modificare l'operazione acquire perchè inserisca i processi che devono attendere in una coda associata al semaforo (che può essere schedulata, ponendo attenzione alla starvation), e la release perché ne estragga uno dalla coda (se ve ne sono). In questo modo, non si elimina del tutto il problema del busy wait (l'accesso alla coda infatti va sincronizzato), ma lo si minimizza, poiché se anche bisogna attendere per inserire un processo in coda perché un altro processo sta venendo inserito, l'attesa durerà poco poiché si tratta di poche istruzioni. Inoltre con i semafori si possono verificare situazioni di attesa circolare (deadlock).
Il problema più grave però è che il loro corretto utilizzo spetta al programmatore, il quale può utilizzarli erroneamente o non utilizzarli affatto, e se ciò accade, ci possono essere violazioni di mutua esclusione, o si possono verificare fenomeni di starvation.
Monitor:
I monitor innalzano ulteriormente il livello di astrazione. Si tratta di un tipo di dato astratto, che, oltre a dei dati, possiede alcuni metodi che sono eseguiti in mutua esclusione. Questa però è gestita a livello di linguaggio di programmazione, in particolare dal compilatore che farà uso delle corrette chiamate di sistema, e quindi non spetta più al programmatore il loro corretto utilizzo.
Ovviamente ciò funziona se si utilizzano i monitor.
Un programmatore può creare un monitor adatto alle proprie esigenze definendo delle variabili di tipo condition, su cui un processo Q può aspettare il verificarsi di una condizione con l'operazione wait. In seguito un processo P potrà eseguire l'operazione signal su una condition, che risveglierà un processo Q dalla coda. A questo punto P può attendere finchè Q lascia il monitor o si mette in attesa di una condizione (signal and wait), oppure viceversa (signal and continue), oppure può lasciare immediatamente il monitor.

Quando più processi condividono tra loro informazioni e le loro computazioni concorrono a uno scopo applicativo comune, si dicono cooperanti. Affinché ciò sia possibile, il SO deve fornire politiche e meccanismi di comunicazione (IPC) e sincronizzazione. Le entità coinvolte nella comunicazione sono il processo mittente (da ora P), il ricevente (da ora Q) e il canale.
Ogni metodo di IPC ha delle caratteristiche proprie. La scelta di un metodo da utilizzare in un'applicazione va fatta in base a:

- Quantità di informazioni da trasmettere
- Velocità di esecuzione
- Scalabilità, ossia se il metodo scelto permette anche a un numero variabile di processi di comunicare
- Semplicità d'uso nelle applicazioni
- Omogeneità delle comunicazioni: è preferibile evitare di utilizzare più metodi di IPC nella stessa applicazione, così da evitare errori comuni
- Integrazione nel linguaggio di programmazione, che garantisce la portabilità
- Affidabilità
- Sicurezza
- Protezione

Si distingue tra:

- Comunicazione diretta, in cui i processi comunicanti conoscono l'ID l'uno dell'altro. Richiede che entrambi i processi siano attivi affinché avvenga la comunicazione
- Comunicazione indiretta, in cui i processi sanno solo dove prelevare/depositare le informazioni

Metodi per la comunicazione diretta:
Memoria condivisa:
E' possibile condividere un'area di memoria tra i processi comunicanti (variabili globali o buffer per comunicazioni).
Si può realizzare in 2 modi:

- Il SO copia la zona di memoria condivisa tra i processi comunicanti. In questo modo i processi comunicano pur rimanendo fisicamente separati. Comporta uno spreco di tempo e memoria per la copia
- Il SO mappa parte dello spazio di indirizzamento logico dei processi sulla stessa area di memoria fisica, che quindi è fisicamente condivisa. Questa tecnica è molto più rapida della precedente e non spreca memoria. I processi rimangono comunque logicamente separati e protetti dal SO

Pur essendo un sistema semplice e veloce, si richiede al programmatore di utilizzare i meccanismi di sincronizzazione poiché i processi possono accedere contemporaneamente agli stessi dati, potenzialmente in conflitto.
Scambio di messaggi:
L'informazione viaggia all'interno di messaggi scambiati dal SO. Sono di lunghezza fissa o variabile e contengono l'informazione da inviare, l'id di mittente e destinatario ed eventuali informazioni di gestione.
I messaggi sono memorizzati in buffer forniti dal SO ad ogni coppia di processi comunicanti, oppure di uso generale tra tutti i processi. È il SO a gestirli, non il programmatore, e non c'è memoria condivisa.
La quantità di buffer può essere:

- Illimitata, P può depositare, teoricamente, infiniti messaggi. In seguito Q potrà riceverli in un qualsiasi momento. Si parla di comunicazione asincrona.
- Limitata, P può depositare messaggi finché ci sono buffer liberi, poi si blocca finché Q non ne libera alcuni. Si parla di comunicazione bufferizzata.
- Nulla, P non può depositare nessun messaggio, e rimane bloccato finché Q non è pronto a ricevere. Vi è quindi un rendezvous tra i processi comunicanti. Si parla di comunicazione sincrona

Le funzioni fornite dal SO sono:

- Send, che invia un messaggio al processo specificato, depositandolo in un buffer libero. Se non ve ne sono, P si blocca finché non può inviare

- Receive, che riceve un messaggio da un processo specificato (caso simmetrico), oppure da un qualsiasi processo che abbia inviato un messaggio a Q (caso asimmetrico). Se non vi sono messaggi ricevibili, Q si blocca finché non ve ne sono

Esistono inoltre le versioni condizionali, che invece di bloccare il processo in attesa, ritornano un codice di errore.
La coda dei messaggi in attesa per un processo può essere schedulata.
Metodi per la comunicazione indiretta:
Mailbox:
Una mailbox è una struttura dati del SO in cui i processi prelevano/depositano messaggi, identificata da un id. La capacità della mailbox può essere illimitata, limitata o nulla. I messaggi sono di dimensione fissa o variabile e contengono l'id di mittente e mailbox destinataria, l'informazione da inviare ed eventuali informazioni di gestione.
Il SO fornisce le seguenti funzioni:

- Create: crea una mailbox
- Send e send condizionale: deposita un messaggio nella mailbox specificata
- Receive e receive condizionale: riceve un messaggio dalla mailbox specificata
- Delete: cancella una mailbox

La mailbox si trova nella memoria del SO, che ne gestisce gli accessi sincronizzandoli. Tipicamente è di sua proprietà, e permette a tutti i processi di utilizzarla. È però possibile assegnare una mailbox a un processo, il quale potrà deciderne i permessi. Quando termina, la mailbox può venire dellocata con esso, oppure può tornare proprietà del SO.
I messaggi nella mailbox possono essere schedulati.
Le mailbox sono particolarmente utili per:

- Comunicazioni molti a 1: un processo di servizio riceve ed esegue le richieste dei client.
- Comunicazioni 1 a molti: un processo client invia richieste ad un qualsiasi processo di servizio disponibile. Il primo a ricevere una richiesta la esegue
- Comunicazioni molti a molti: più processi client inviano richieste a un qualsiasi processo di servizio disponibile. Il primo a ricevere una richiesta la esegue

Ogni comunicazione comunque, coinvolge solo 2 processi: P e Q.
Comunicazione tramite file e pipe:
I file sono utili per scambiare grandi quantità di dati: i messaggi sono depositati e prelevati da un file su memoria di massa. Essendo su memoria di massa, la comunicazione è piuttosto lenta. Gli accessi sono sincronizzati dal file system. L'ordinamento dei messaggi dipende dal processo scrivente.
Una pipe può essere vista come un file memorizzato in memoria del SO. Le funzioni per utilizzarle sono le stesse che si usano per i file, e la sincronizzazione avviene allo stesso modo. L'ordinamento dei messaggi è FIFO.
Sia nei file che nelle pipe i messaggi sono di lunghezza fissa o variabile e contengono l'id del mittente, l'informazione da trasmettere ed eventuali informazioni di gestione.
Comunicazione tramite socket:
I socket sono utilizzati per le comunicazioni in rete, ma possono essere usati anche per l'IPC, anche tra processi su macchine separate: 1 o più mittenti si collegano al ricevente specificandone l'indirizzo e la porta su cui è in ascolto. Una volta connessi, avviene la comunicazione.
Un socket fornisce ai processi un canale bidirezionale che può essere visto come 2 code con i versi opposti spezzate al centro, con gli estremi sulle 2 macchine comunicanti.
I messaggi sono di dimensione fissa o variabile e contengono solo l'informazione da scambiare. Sono in ordine FIFO.

Il deadlock (o stallo) è una condizione in cui ogni processo in un gruppo è bloccato in attesa di un evento che può essere generato solo da un altro processo del gruppo, tipicamente il rilascio di risorse fisiche o logiche.
Affinchè il deadlock si possa verificare devono essere soddifatte contemporaneamente 4 condizioni:

- Mutua esclusione: almeno una delle risorse coinvolte è utilizzata in modo mutualmente esclusivo dai processi
- Possesso e attesa: un processo che possiede delle risorse si mette in attesa per averne altre
- Nessun rilascio anticipato: il sistema operativo non può forzare il rilascio di risorse assegnate ai processi
- Attesa circolare: n processi attendono in modo che P1 attende P2, P2 attende P3, ..., Pn attende P1

Vediamo ora come si può affrontare il problema.
Ignorare il deadlock:
Nella maggior parte dei sistemi, il deadlock è una condizione rara e quindi si può risparmiare tempo e aumentare l'uso della risorse ignorandolo e resettando il sistema se si verifica.
Prevenire il deadlock:
Si agisce invalidando una delle 4 condizioni che causano il deadlock imponendo dei vincoli sulle richieste, che però comportano un potenziale sottoutilizzo delle risorse e sono scomodi per il programmatore.
Mutua esclusione:
Alcune risorse sono intrinsecamente condivisibili (ad esempio file in lettura), e per queste si può invalidare; tuttavia, in generale non è possibile farlo poiché non tutte lo sono.
Possesso e attesa:
Si può invalidare imponendo che i processi richiedano tutte le risorse all'avvio, oppure imponendo che un processo, prima di fare una richiesta, rilasci tutte le risorse che possiede e le richieda tutte in blocco. In entrambi i casi si ha un sottoutilizzo delle risorse e possibile starvation; inoltre, nel secondo caso, si richiede ai programmatori di rilasciare le risorse in stato consistente poiché potrebbero essere assegnate ad altri processi.
Nessun rilascio anticipato:
Si può invalidarla in 2 modi: quando un processo viene messo in attesa per una richiesta, tutte le sue risorse sono rilasciate anticipatamente, e quindi attende per la richiesta più le risorse che già possedeva. In alternativa, si possono rilasciare anticipatamente solo le risorse di processi in attesa che sono richieste da un processo in esecuzione.
Il problema della consistenza descritto prima rimane, tuttavia è il sistema operativo che rilascia le risorse, e non il programmatore.
Questo metodo funziona bene con risorse il cui stato può essere salvato e ripristinato.
Attesa circolare:
Si può invalidare imponendo un ordinamento sulle risorse, ossia una definendo una funzione che assegni ad ogni risorsa un numero intero, e imponendo questo protocollo: quando un processo richiede risorse di ordine i, questi può ottenerle solo se sono disponibili e non possiede già risorse di ordine >=i. Se ne possiede, le deve rilasciare e richiedere in ordine corretto (con tutti i problemi relativi). In questo modo, l'attesa circolare non può verificarsi, ma la funzione va definita in base ai consueti ordini di utilizzo delle risorse, e può non essere facile.
Evitare il deadlock:
I metodi per evitare il deadlock non pongono vincoli su come fare le richieste, ma chiedono al processo informazioni supplementari sull'utilizzo delle risorse (quali, quante, in che ordine, ...), e le utilizzano per verificare se lo stato risultante da una richiesta è sicuro. Il problema è che queste informazioni potrebbero non essere note a priori.
Uno stato si dice sicuro per n processi se esiste almeno una sequenza sicura, ossia una sequenza di n processi per cui le richieste di ogni processo Pi della sequenza possono essere soddisfatte con le risorse attualmente disponibili più quelle detenute dai processi Pj, con j<i (poiché le rilasceranno). Se non esiste una tale sequenza, lo stato si dice non sicuro. Uno stato sicuro garantisce l'assenza di deadlock,

ma uno stato non sicuro non significa necessariamente deadlock.
Verranno ora presentati 2 metodi che richiedono di sapere quali e quante risorse saranno usate da un processo.
In caso di istanze singole delle risorse, si può utilizzare una variante del grafo di allocazione delle risorse. Quest'ultimo è un grafo orientato che ha come vertici i processi e le risorse, e come archi ha archi di assegnazione (da una risorsa a un processo) e archi di richiesta (da un processo a una risorsa). Questa variante introduce gli archi di prenotazione (da un processo a una risorsa). All'avvio, un processo deve specificare quali risorse utilizzerà, così da generare i suoi archi di prenotazione. Quando richiederà una risorsa (disponibile), si controlla se la conversione del relativo arco di prenotazione ad arco di assegnazione causa cicli nel grafo. Se ne causa, lo stato non è sicuro e il processo deve attendere.
In caso di istanze multiple si può utilizzare il più complesso algoritmo del banchiere.
Fa uso di alcune strutture dati per mantenere le informazioni sui processi(n= numero processi, m=numero risorse):

- Available: array di m elementi contenente il numero di istanze disponibili per ogni risorsa
- Allocation: matrice n*m contenente le allocazioni per ogni processo
- Max: matrice n*m contenente per ogni processo le risorse massime che può utilizzare
- Need: matrice n*m contenente per ogni processo le risorse massime che può ancora richiedere

Quando un processo fa una richiesta, si controlla se è valida (<= need di quel processo) e se è soddisfacibile (<= available), quindi modifica le strutture dati per simulare l'assegnazione della risorsa, ed esegue l'algoritmo di verifica dello stato sicuro: si cercano tutti i processi il cui fabbisogno (need) può essere soddisfatto con le attuali disponibilità più le risorse assegnate ai processi analizzati precedentemente (che le rilasceranno al termine). Se alcuni processi rimangono scoperti, lo stato non è sicuro, si ripristinano i vecchi valori delle strutture dati e il processo deve attendere. In caso contrario, il processo ottiene le risorse richieste.
Rilevamento e risoluzione del deadlock:
Un ultimo approccio che consente di aumentare l'utilizzo delle risorse è lasciare che il deadlock si verifichi, e usare degli algoritmi per rilevarlo e risolverlo.
In caso di istanze singole delle risorse, si può usare il grafo di attesa: una variante del grafo di allocazione con i nodi risorsa collassati per mostrare le attese tra i processi. Se nel grafo ci sono cicli, quei processi sono coinvolti in un deadlock.
In caso di istanze multiple si può usare un algoritmo simile all'algoritmo di verifica dell'algoritmo del banchiere. Utilizza le stesse strutture dati (tranne need e max, poiché non sono note), più la matrice request n*m che contiene le attese di ogni processo. Si cercano tutti i processi le cui richieste possono essere soddisfatte con le attuali disponibilità più le risorse detenute dai processi analizzati precedentemente (si suppone che terminino e le rilascino, se non lo fanno, il deadlock verrà rilevato successivamente). Se alcuni processi rimangono scoperti, questi sono coinvolti in un deadlock.
Per risolverlo si possono terminare tutti i processi coinvolti (viene risolto, ma si possono perdere i risultati di lunghe computazioni), oppure terminarli uno per volta in base a qualche criterio finché non si risolve il deadlock (evitando di terminare sempre gli stessi causandone la starvation).
L'algoritmo di rilevamento può essere eseguito ogni n secondi, oppure quando l'utilizzo della cpu scende sotto una certa soglia (se c'è un deadlock, sempre più processi saranno coinvolti quindi continuerà a scendere).