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.