I Thread hanno delle risorse proprie :

- Stack ;
- PC ;
- Stato ;
- Registri ;

mentre invece condividono :

- Memoria ;
- File aperti ;
- Codice ;

La gestione degli stati dei thread è quindi similare a quella dei processi. Una singola applicazione, se scritta su più thread, può avere vantaggi sulla scalabilità e sulle capacità computazionali . Il modello è “non bloccante” in quanto i thread agiscono in maniera indipendente tra loro e l’attesa di un thread non blocca l’esecuzione dell’intero processo ma si porta avanti un thread che può proseguire con la propria esecuzione.

La programmazione multi-thread ha un costo in termini di complessità dell’applicazione. Un approccio diverso è dato dalla programmazione multi-core.
Il primo step è individuare e separare in modo netto , quali sono i TASK delle varie applicazioni :

#1

es un task per la GUI, uno per il correttore ortografico, ecc..

Nel fare questa ricerca si tiene conto di alcuni criteri : non tutto deve essere visto come un task ma bisogna distribuire il giusto carico tra i vari task in termini di carico di lavoro , di vita del task ecc..

#2

#3

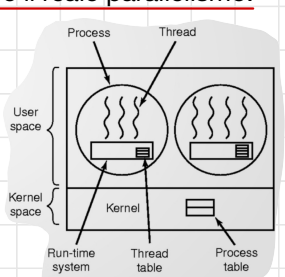
Un’altra operazione è quella di gestire la suddivisione dei dati, si devono individuare le dipendenze dei vari task attraverso un processo di delineamento delle dipendenze dei dati condivisi (file, ecc..) e tutto ciò verrà gestito tramite meccanismi di sincronizzazione . La gestione di ogni task viene associata di norma ad un thread. Segue una fase di testing e debugging. La complicazione di questa fase viene data dalla scarsa linearità derivante dalla gestione in parallelo di diversi flussi. Il manifestarsi di un anomalia implica che si deve trovare un bug dovuto all’interlacciamento tra varie operazioni differenti. In questo caso si deve forzare le operazioni ad interlacciarsi nel modo desiderato.

#4

#5

Thread a livello utente —> modello 1-a-molti

Nei primi modelli architetturali della programmazione multi-thread è la seguente : la gestione in diversi thread del processo anche in assenza del supporto del kernel che , essendo pensato per gestire solo processi (quindi con un solo thread), non ci mette nulla di suo per aiutare la programmazione. A livello utente (con i limiti del caso) si necessita di implementare le relative operazioni tra thread e le relative strutture dati a livello di codice nello spazio di indirizzamento del processo. L’idea è quella di accorpare più processi di un singolo thread all’interno di un singolo thread. Entra in gioco molto la thread_yield per cedere la CPU in maniera volontaria ed indipendente dalla CPU : il SO salva i registri del thread che cede la CPU all’interno dei registri nella thread table usando operazioni ordinarie in UserMode e gestisce un problema di scheduling limitato ai vari thread che possono essere mandati in esecuzione. Viene simulato quindi il comportamento del SO implementando del codice in User Mode.
Si sfruttano le liberie di Run time Environment vengono sfruttate dal programmatore per implementare i thread. Il sistema operativo è costretto ad assegnare l’intero processo ad una sola CPU su un singolo Core e non sarà possibile sfruttare la scalabilità e il reale parallelismo.



Uno dei principali problemi di questo modello è che se un thread viene bloccato da una chiamata lenta , non si potrà sfruttare l'indipendenza dei thread per passare all'esecuzione di un altro thread. Una prima strategia per mitigare questo problema è quella di aggirare la chiamata bloccante, attraverso chiamate di sistema di tipo Select per capire prima di lanciarla se l'istruzione potrà essere bloccante : ad esempio se la richiesta è un socket di rete che potrebbe bloccare se non riceve il pacchetto, allora guardo a priori attraverso la Select se la chiamata potrebbe bloccare e la inoltro solo se l'ispezione preliminare mi garantisce che non sarà generato un blocco. Se la select mi dice che l'istruzione potrebbe essere bloccate (caso del socket) si invoca la thread_yield() per cedere la CPU ad un altro thread che può proseguire la propria esecuzione tranquillamente. Questo modo per affrontare il problema introduce ovviamente uno spreco della CPU simile a quello introdotto dalla busy waiting, inoltre si deve modificare il codice per introdurre delle istruzioni che normalmente non ci sarebbero. Si necessita di una collaborazione tra i thread per evitare che un processo monopolizzi la CPU.

Il page-fault è un evento dovuto al fatto che gestendo la memoria centrale, non si trova il dato desiderato in RAM e si necessita che il Sistema Operativo fa una segnalazione (detta page fault) all'hardware perchè l'istruzione che ha richiesto il fetch di quel dato non può essere eseguita e quindi si crea il blocco dovuto al fatto che la word deve essere prelevata dal disco. Quando la gestione del page fault è completa , l'istruzione che aveva richiesto il dato viene ripetuta e stavolta andrà a buon fine. Questo evento è un blocco asincrono in quanto non è dovuta ad una system call effettuata dal codice ma all'organizzazione in memoria dei dati.

Questo modello introduce un ridotto overhead nella gestione dei thread , è più efficiente e leggero in quanto la thread_yield() implementa il context switch tra i vari thread e viene fatto tutto a livello utente, non abbiamo chiamate di sistema . Questo è il context switch più veloce dovuto al fatto che non si interpella il kernel tramite le sistem call, non si esce mai dalla user mode.

1) Context-Switch tra thread fratelli nell'implementazione a livello utente —> più veloci
2) Context-Switch tra thread fratelli (thread imparenti)
3) Context-Switch tra processi (thread non imparentati) —> più lenti

Thread a livello kernel —> modello 1-a-1

Qui il sistema operativo conosce i thread e quando un'applicazione necessita di coordinare i thread, viene supportato dal Kernel e viene implementato praticamente su tutti i sistemi moderni. Si ha un'unica tabella dei thread del kernel in modo da garantire una retro compatibilità con i modelli dove i thread erano a livello utente. Qui abbiamo il vantaggio di poter mandare avanti l'esecuzione di altri thread qualora quello in esecuzione dovesse bloccarsi, garantendo una maggiore scalabilità del sistema. Al posto di creare e distruggere processi, si tiene conto che queste operazioni hanno un costo, di sfrutta il modello a workers :
un thread si incarica di ricevere la richiesta e la smista ai vari workers che svolgono il loro compito e vengono poi riaddormentati. Si ha un numero minimo prestabilito di thread-workers che sono incaricabili di un lavoro.
Il cambio di contesto richiede l'intervento del kernel dovuto alle sistem call e quindi si ha un maggiore overhead che rallentano il context-switch.

Modello ibrido —> molti-a-molti

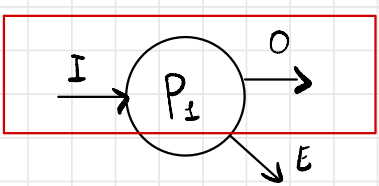
Nei precedenti modelli vengono introdotti pro e contro ma si può ricorrere alla creazione di un modello ibrido che massimizzi i pro e lenisca i contro. Si sfruttano i thread multipli all'interno del kernel e si individuano un certo numero di task all'interno dell'applicazione . Si pensa di allocare un solo thread kernel che viene sfruttato per gestire molti thread-task. Si associa la gestione di molti task a diversi thread, adesso il vantaggio è che all'interno dello stesso ambiente abbiamo molti task potrebbero non avere la necessità di essere allocati all'interno di un solo thread e quindi non si ha il costo di creare un nuovo thread per la gestione di tasks che potrebbero essere gestiti da un unico thread. In questo caso si rende accettabile che il bloccarsi di un thread blocchi anche altri thread. Si sfrutta quindi il modello a livello utente in un sottoscenario del modello kernel.

Ad esempio implementare il correttore ortografico e la gestione di I/O all'interno dello stesso thread_kernel comporta che se si blocca il processo di gestione di I/O venga bloccato anche il correttore ortografico, il che non è un problema per l'utilizzatore.

Comunicazione tra processi (inter process communication IPC)

I processi sono visti come isolati e normalmente non posso uscire dal proprio spazio di indirizzamento, però potrebbe nascere l'esigenza di mettere in comunicazione processi diversi. Ogni processo, non appena creato, ha dei canali di comunicazione standard :

- Input : legato alla tastiera
- Output : legato al terminale stesso, così come anche il messaggio di errore
- Errori : pensato per messaggi di servizio e warning , differenziato da quello di output per tenere pulito l'output



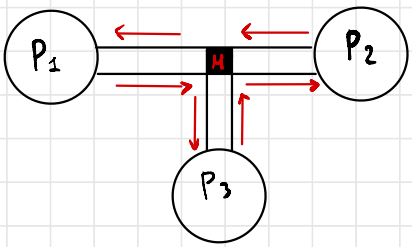
$ cat lista.txt | grep parola | sort

A livello di processi succede che la shell crea 3 fork e fa si che siano collegati, in quanto l'output del primo comando sarà preso in input dal secondo e così via. Questo è un esempio di interprocess communication . In realtà i tre processi agiscono in parallelo ma si ha un buffer intermedio che passa l'output man mano che viene prodotto al comando successivo in quanto gli altri processi si bloccheranno se non hanno nulla in input. Si potrebbe verificare una situazione per la quale il processo 2 dorme e il processo 1 si deve bloccare perchè non può mandare in output un dato che non verrà "consumato" (modello produttore-consumatore). Allo stesso modo se il comando 1 termina , il canale di comunicazione viene chiuso ed il comando 2 non avrebbe nulla da leggere.

La comunicazione quindi è osservabile come un flusso di dati in questo modello semplice da usare. L'utilizzo di modelli di comunicazione più generali permette una più efficiente comunicazione :

- se non utilizzo il canale (troppo rigido) come comunicano i processi , essendo isolati?

L'idea è di avere una finestra di memoria comune ai processi a cui hanno accesso in tempo reale e che consente lo scambio di dati.



Race condition

Se utilizzo i thread posso sfruttare lo spazio di indirizzamento condiviso per comunicare, si necessità però di una gestione della temporizzazione.
La race condition è un problema comune che può verificarsi quando due o più thread di un programma accedono a una risorsa condivisa, come una variabile o un file, allo stesso tempo e senza un'adeguata sincronizzazione. In questo scenario, l'output o il comportamento del programma diventa dipendente dalla sequenza temporale degli accessi ai dati, che possono essere imprevedibili e quindi portare a comportamenti errati. Il termine "race condition" deriva dal fatto che i thread "gareggiano" per accedere alla risorsa condivisa, e il risultato dipende da quale thread arriva per primo. La race condition si può verificare anche per accesso alle strutture dati interne al kernel, quindi questo è un problema serio che può manifestarsi a causa dell'accavallamento degli utilizzi a qualsiasi livello.
La soluzione consiste nella mutua esclusione nell'accesso dei dati condivisi, ovvero si garantisce che quando uno dei processi accede a un dato lo fa in modo esclusivo e garantendosi che nessun altro processo acceda alla propria sezione critica.